

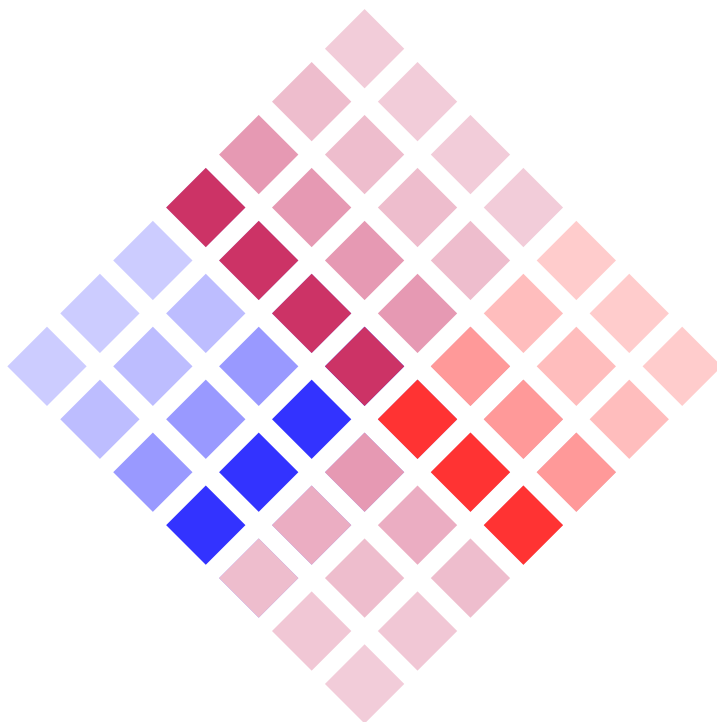
Satisfiability of Short Circuit Logic

Sander in 't Veld

October 20, 2015

Bachelor's Thesis in Mathematics and Computer Science (revised)

Supervisors: dr. Inge Bethke, prof. dr. Jan van Eijck



Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam



Abstract

The logical connectives typically found in programming languages are similar to their mathematical counterparts, yet different due to their short-circuit behaviour – when evaluating them, the second argument is only evaluated if the first argument is not sufficient to determine the result. Combined with the possibility of side-effects, this creates a different type of logic called Short Circuit Logic. A greater theoretical understanding of this logic can lead to more efficient programming and faster program execution.

In this thesis, formula satisfiability in the context of Short Circuit Logic is discussed. A formal definition of evaluation based on valuation algebras is presented, alongside an alternative definition based on valuation paths. The accompanying satisfiability and ‘path-satisfiability’ are then proven to be equivalent, and an implementation of path-satisfiability is given. Although five types of valuation algebras can be discerned, there are only three corresponding types of valuation paths. From this, conclusions are drawn about satisfiability and side-effects; the manner in which side-effects alter truth values is relevant when analysing satisfiability, but the side-effects themselves are not.

Title: Satisfiability of Short Circuit Logic

Authors: Sander in 't Veld, sander.intveld@student.uva.nl

Supervisors: dr. Inge Bethke, prof. dr. Jan van Eijck

Date: October 20, 2015

Universiteit van Amsterdam

Science Park 904, 1098 XH Amsterdam

<http://www.science.uva.nl>

Contents

1. Introduction	5
2. Preliminaries	7
2.1. Notation	7
2.2. Formulas	7
2.3. Short Circuit Logics	8
2.4. Evaluation Trees	9
2.5. Normal Form	11
3. Evaluation and Satisfiability	13
3.1. Valuation Algebras	13
3.2. Satisfiability	19
4. Path-Satisfiability	21
4.1. Valuation Paths	21
4.2. Path-Satisfiability	25
4.3. Norm-based Constructors	29
4.4. Satisfiability and Path-Satisfiability	34
5. Implementation in Haskell	39
5.1. Formulas, Trees and Paths	39
5.2. Satisfiability Testers	41
6. Conclusion	47
Bibliography	48
A. Axioms of Short Circuit Logics	49
B. Proof of Proposition 3.9	51

1. Introduction

The field of logic deals with formulas and truths. In propositional logic, a formula containing proposition letters p and q is said to be *satisfiable* if each of the letters can be assigned a value, either true or false, such that the formula as a whole becomes true. For example, the formula $p \wedge \neg q$, which is read as “ p and not q ”, is satisfiable by taking p to be true and q to be false. On the other hand, the formula $p \wedge \neg p$ is not satisfiable in propositional logic, as p cannot be simultaneously true and false.

Consider the following code fragment, written in C-like pseudocode.

```
integer n = 0
boolean a() { ... }
boolean b() { ... }

if ( a() && b() && !a() )
{
    print("Hello")
}
```

We have one integer variable `n` and two functions `a()` and `b()` that take no arguments and return booleans. Whether or not ‘Hello’ is printed only depends on the value of `n`. However, it is possible that nothing will ever be printed, no matter what value we choose. For instance, if `a()` simply always returns true, then `!a()` will always be false, and the line `print("Hello")` will never be reached. In this case, `print("Hello")` is a piece of “dead code”. Being able to detect dead code is of great interest to compilers and optimisers, as the dead code is often the result of an error by the programmer, and since removing it reduces memory and cpu usage. If we translate the if-clause `a() && b() && !a()` to the logical formula $a \wedge b \wedge \neg a$, then detecting dead code is similar to answering the question “Is this formula satisfiable?”. This is one of the many reasons why logicians and computer scientists seek a greater understanding of satisfiability.

When evaluating the formula $x \wedge y$, we usually first evaluate x and y separately. Then $x \wedge y$ is true if both x and y are true, and it is false if at least one of x and y is false. However, if x is false then knowing this is enough to determine that $x \wedge y$ must also be false; the value of y no longer needs to be considered. Computer programs can make use of this fact in what is called *short-circuit evaluation*.

Common programming languages such as C, Java and Haskell feature short-circuit evaluation in the form of the logical connectives `&&` and `||`. A typical example of an expression using such a connective is

$$(n \neq 0) \ \&\& \ (x/n < 1)$$

where `n` and `x` are integer variables. Here the right-hand side of the expression, which features a relatively expensive division operation, will only be evaluated if the left-hand side evaluates to true. Besides being expensive, the division operation comes with the danger of ‘division by zero’, which will result in a program crash on most platforms. Short-circuit evaluation in this case ensures that the expression will always return a value, as expected.

Also of relevance to logic in computer programs are *side-effects*; the evaluation of a formula might change the state of the context in which it is evaluated. ‘Division by zero’ could be considered an example of this, but its effect is so drastic that we will not further discuss it here. Instead, the assignment operator `=` as found in the C language provides a better example. The expression `(n = 55)` will assign the value 55 to `n` and return true. Clearly, the evaluation of such an expression will affect the evaluation of later expressions containing `x`.

Detecting dead code is *similar* to solving propositional satisfiability, but not the same. In propositional logic, the formula $p \wedge q \wedge \neg p$ is unsatisfiable, but if we fill in the functions `a()` and `b()` from our code fragment as

```
boolean a() { return (n == 0) }
boolean b() { return (n = 55) }
```

then the program would print ‘Hello’. Thus, short-circuit evaluation and side-effects appear to be part of a different kind of logic.

In *Short Circuit Logic* [1], the semantics of short-circuit evaluation and side-effects are described in more detail. A new type of logic, the short-circuit logic, is introduced, and the logics FSCL, RPSCL, CSCL, MSCL and SSCL are defined and axiomatised.

This thesis attempts to formally define what evaluation and satisfiability mean in the context of Short Circuit Logic, and suggests and implements a few methods to test the satisfiability of a formula with regards to these five logics. Relevant questions are:

- ▷ *How does satisfiability for Short Circuit Logic differ from traditional satisfiability?*
- ▷ *How do different types of side-effect change satisfiability?*
- ▷ *Can short-circuit evaluation be utilised while testing satisfiability?*

The next chapter will be spent laying the groundwork, as well as summarizing a few results from [1]. In Chapter 3, we will formally define evaluation and satisfiability for Short Circuit Logic. An implementation for testing satisfiability will be discussed in Chapter 5, but it will at first seem incompatible with the definitions from Chapter 3. The gap between theory and implementation will be bridged in Chapter 4, where we will define an alternative definition of satisfiability. Finally, Chapter 6 will reconsider the questions asked above.

2. Preliminaries

2.1. Notation

Throughout this thesis, we will consider the left-sequential short-circuit versions of the connectives \wedge and \vee used in traditional logic. Here ‘left-sequential’ means that the left-hand side is evaluated before the right-hand side, and short-circuit means that the right-hand side is only evaluated if the left-hand side is not enough to determine the result. We will follow notation featured in [5] and [1] and use the symbols \Join and \JoinV for these connectives. Additionally, the symbols T and F will be used for the truth values ‘true’ and ‘false’ respectively, and the symbol \neg for logical negation. Furthermore, the symbols \sqsubseteq and \sqsupseteq will be used to describe certain binary trees.

In earlier work, the connectives \Join and \JoinV are defined based on Hoare’s conditional, $_ \triangleleft _ \triangleright _$. In this thesis, we are not specifically interested in this conditional, and directly use the results from these works.

2.2. Formulas

Whereas propositional logic considers formulas over a certain set Φ of proposition letters, Short Circuit Logic considers formulas over a set \mathcal{A} of *atoms*. The intuitive difference between proposition letters and atoms is that atoms can have side-effects. Throughout this thesis we will assume we have fixed a set \mathcal{A} of atoms. The formulas of Short Circuit Logic are given by a few basic rules. First, the constants T and F are formulas, and each atom $a \in \mathcal{A}$ is a formula as well. Furthermore, if x and y are formulas, then so are $\neg x$, $x \Join y$ and $x \JoinV y$. More formally:

Definition 2.1. The *formulas* over \mathcal{A} are defined by the following grammar:

$$x ::= \text{T} \mid a \mid \neg x \mid x \Join x$$

where a ranges over \mathcal{A} .

The two symbols F and \JoinV seem to be missing from the above definition. Although adding them is possible, it would make induction proofs slightly less practical. Therefore, as is not uncommon in other fields of logic, we define F and \JoinV as abbreviations:

$$\text{F} := \neg \text{T}, \quad x \JoinV y := \neg(\neg x \Join \neg y).$$

We need brackets to indicate precedence in more complicated formulas. As an example, $\neg(x \Join y)$ is the negation of $x \Join y$, whereas $\neg x \Join y$ is the conjunction of $\neg x$ and y .

Throughout this thesis, we will make repeated use of induction to the complexity of formulas and other objects. This can be formalised with an adequate definition of complexity, such as the following:

Definition 2.2. Let x be a formula. The *complexity* of x is defined recursively:

$$\begin{aligned} \text{cx}(\text{T}) &= 0, \\ \text{cx}(a) &= 0, \quad \text{for each } a \in \mathcal{A}, \\ \text{cx}(\neg x_1) &= 1 + \text{cx}(x_1), \\ \text{cx}(x_1 \frown x_2) &= 1 + \max\{\text{cx}(x_1), \text{cx}(x_2)\}. \end{aligned}$$

However, we will usually just remark that a proof is by induction and omit any formal inductive structure, for the sake of brevity. Lastly, we define what it means for a formula to be ‘constant-free’.

Definition 2.3. A formula is called *constant-free* if it contains neither T nor F, i.e. if it is defined by the following grammar:

$$x ::= a \mid \neg x \mid x \frown x$$

where a ranges over \mathcal{A} .

2.3. Short Circuit Logics

Logics identify certain formulas. That is, if x and y are formulas, then some logics might consider x and y to be ‘the same’; not in the sense of their structure or complexity, but in the way that they behave as formulas. For instance, the formulas T and $\neg\text{F}$ are very different in appearance, but both have the same semantical interpretation: ‘true’. If x and y are identified formulas, then so are $\neg x$ and $\neg y$, as well as $x \frown z$ and $y \frown z$ for any z , etcetera.

In [1] and [3], five short-circuit logics are introduced: FSCL, RPSCL, CSCL, MSCL and SSCL. The names are abbreviations of “free –”, “repetition-proof –”, “contractive –”, “memorizing –” and “static short-circuit logic” respectively. We will not go in detail about their definitions, but instead briefly discuss the intuitive differences between the five logics.

The logic FSCL is the least identifying short-circuit logic. As such, this logic describes only the most fundamental properties of the symbols T, \neg and \frown . This logic allows all types of side-effects.

In RPSCL, atoms must retain their value when evaluated multiple times in a row; that is, if a is true, then $a \frown a$ must also be true. The logic CSCL takes this a bit further and demands that only the first evaluation of two identical atoms can have a side-effect. Thus, in CSCL, if $a \frown b$ is true, then so is $a \frown a \frown b$, because the second occurrence of a cannot have a side-effect that makes b false. In MSCL, the effects and values of atoms are ‘memorised’ entirely. This means that once a have been evaluated to true, any further evaluations of a must also lead to true and can have no further side-effects.

The logic SSCL is the most identifying and restrictive short-circuit logic. In this logic, there are no side-effects; or rather, the side-effect of an atom cannot actually affect what values later atoms will take. As such, the logic SSCL is equivalent to propositional logic. This means that if we take a formula in SSCL and replace every \top by \top , every Δ by \wedge , every atom $a \in \mathcal{A}$ by a corresponding proposition letter $p \in \Phi$, etcetera, then evaluating the formula in SSCL is the same as assigning either ‘true’ or ‘false’ to each of the proposition letters in the translation.

If E is an axiom system, i.e. a collection of axioms, then we write $E \vdash x = y$ if the logical statement “ $x = y$ ” can be proven by using axioms from E and logical tautologies. An axiom system is *sound* for a logic if every two formulas that are proven equal by the axioms, are identified by the logic. An axiom system is *complete* for a logic if every two formulas that are identified by the logic, can be proven equal by the axioms. If an axiom system is both sound and complete for a certain logic, then it axiomatises this logic.

The logic FSCL is axiomatised by the system EqFSCL, while RPSCL is axiomatised by EqRPSCL, etcetera. These axiom systems can be found in the appendix. The soundness and completeness of each of the respective axiom systems is discussed in [1].

2.4. Evaluation Trees

Binary trees are one of the most simple ways to emulate choice: starting at the root of a tree, we can go down either the left or the right branch. Once we have gone down either, we may encounter another choice, and after that yet more choices, until we eventually arrive at a ‘leaf’, where the journey down the tree ends. In Short Circuit Logic, we are interested in a specific type of trees.

Definition 2.4. The *trees* over \mathcal{A} are defined by the following grammar:

$$X ::= T \mid F \mid X \leq a \geq X$$

where a ranges over \mathcal{A} .

Figure 2.1 depicts the tree $(F \leq b \geq T) \leq a \geq F$. In these trees, the ‘choices’ are atoms from our set \mathcal{A} , and our leaves are truth values. The supposed meaning is this: starting at the root, we encounter the atom a . If a is true, then we descend down the left branch and encounter another atom: b . However, if a is false, we take the right branch and we immediately arrive at a leaf: F . This is reminiscent of the short-circuit behaviour we are looking for.

To allow us define trees recursively, we will use substitution. Suppose we have a tree X and we want to somehow ‘extend’ this tree, then we can do this by replacing each of its leaves by new trees. Formally, we define a *substitution* as follows:

Definition 2.5. Let X , Y and Z be trees. We define $X[T \mapsto Y, F \mapsto Z]$ as:

$$\begin{aligned} T[T \mapsto Y, F \mapsto Z] &= Y \\ F[T \mapsto Y, F \mapsto Z] &= Z \\ (X_1 \leq a \geq X_2)[T \mapsto Y, F \mapsto Z] &= X_1[T \mapsto Y, F \mapsto Z] \leq a \geq X_2[T \mapsto Y, F \mapsto Z] \end{aligned}$$

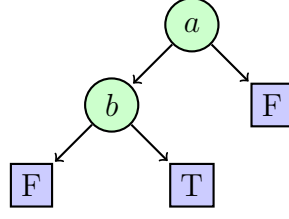


Figure 2.1.: A graphical depiction of the tree $(F \trianglelefteq b \trianglerighteq T) \trianglelefteq a \trianglerighteq F$.

Thus, if X , Y and Z are trees, then $X[T \mapsto Y, F \mapsto Z]$ is the tree X where each T leaf is replaced by Y and each F leaf by Z . As an example, the tree in Figure 2.1 can also be written as $(T \trianglelefteq a \trianglerighteq F)[T \mapsto (F \trianglelefteq b \trianglerighteq T), F \mapsto F]$. Additionally, note that the substitution $[T \mapsto T, F \mapsto F]$ does not alter trees, whereas $[T \mapsto F, F \mapsto T]$ simply swaps the T and F leaves.

The real significance of trees is given by the following definition:

Definition 2.6. The *short-circuit evaluation tree* of a formula x , denoted $\text{se}(x)$, is defined as follows:

$$\begin{aligned} \text{se}(T) &= T \\ \text{se}(a) &= T \trianglelefteq a \trianglerighteq F \\ \text{se}(\neg x) &= \text{se}(x)[T \mapsto F, F \mapsto T] \\ \text{se}(x \wp y) &= \text{se}(x)[T \mapsto \text{se}(y), F \mapsto F] \end{aligned}$$

Remark. The following equalities can be derived:

$$\begin{aligned} \text{se}(F) &= F \\ \text{se}(x \vee y) &= \text{se}(x)[T \mapsto T, F \mapsto \text{se}(y)] \end{aligned}$$

They are not part of the definition, as F and \vee are abbreviations.

The tree depicted in Figure 2.1 is in fact the se -tree of $a \wp \neg b$. Note that as the atom a appears before atom b in the formula $a \wp \neg b$, it also appears earlier (i.e. higher) in the tree. However, not all atoms from a formula necessarily appear in the tree, as is apparent from $\text{se}(T \vee a) = T$. Still, se -trees exactly represent the ‘behaviour’ of formulas. This fact is proven in [1] by Theorems 2.1.7, 3.2.2 and 3.5.2, and summarised as the following theorem:

Theorem 2.7. *If x and y are formulas, then $\text{EqFSCL} \vdash x = y \iff \text{se}(x) = \text{se}(y)$.*

We will need a few more definitions throughout the following chapters, along with a small proposition.

Definition 2.8. Let X be a tree. The *depth* of X is defined recursively:

$$\begin{aligned} \text{depth}(T) &= \text{depth}(F) = 0 \\ \text{depth}(X_1 \trianglelefteq a \trianglerighteq X_2) &= 1 + \max\{\text{depth}(X_1), \text{depth}(X_2)\}. \end{aligned}$$

Definition 2.9. A tree is called *closed* by T or F if all of its leaves are T or F respectively. A tree is called *open* if it is not closed.

Proposition 2.10. *Let X, Y and Z be trees. If X is open and at least one of Y and Z is open, then $X[T \mapsto Y, F \mapsto Z]$ is open.*

Proof. Let X, Y and Z be trees such that X and at least one of Y and Z is open. Suppose Y is open. Because X is open, it contains at least one T leaf. In $X[T \mapsto Y, F \mapsto Z]$, this leaf is replaced by Y , and therefore this new tree is open because Y is open. If Y is not open, Z must be open. Because X is open, it also contains at least one F leaf, which is replaced by Z in the new tree, and now $X[T \mapsto Y, F \mapsto Z]$ is open because Z is open. \square

Corollary 2.11. *If x is a constant-free formula, then $\text{se}(x)$ is open.*

Proof. Of course T is not constant-free. Clearly $\text{se}(a)$ is open for all $a \in \mathcal{A}$. Let $\neg x$ be constant-free, then so is x . By induction we may assume that this means $\text{se}(x)$ is open, and therefore $\text{se}(\neg x) = \text{se}(x)[T \mapsto F, F \mapsto T]$ is also open. Let $x \triangleleft y$ be constant-free, then so are x and y , thus $\text{se}(x)$ and $\text{se}(y)$ are open. Now Proposition 2.10 tells us that $\text{se}(x \triangleleft y)$ is also open. By induction, every constant-free formula has an open se-tree. \square

2.5. Normal Form

One final preliminary is the *normal form*. This type of formula bridges the gap between formulas and se-trees. Its definition is slightly more complex and is justified in [1].

Definition 2.12. Consider the following grammar, where a ranges over \mathcal{A} :

$$\begin{aligned} P &::= P^T \mid P^F \mid P^T \triangleleft P^* \\ P^T &::= T \mid (a \triangleleft P^T) \vee P^T \\ P^F &::= F \mid (a \vee P^F) \triangleleft P^F \\ P^* &::= P^c \mid P^d \\ P^\ell &::= (a \triangleleft P^T) \vee P^F \mid (\neg a \triangleleft P^T) \vee P^F \\ P^c &::= P^\ell \mid P^* \triangleleft P^d \\ P^d &::= P^\ell \mid P^* \vee P^c \end{aligned}$$

A formula is in *normal form* if it is defined by P in this grammar. The formulas defined by P^T are known as T-terms; P^F defines F-terms, P^ℓ defines ℓ -terms and P^* defines *-terms. The formulas of the form $P^T \triangleleft P^*$ are known as T*-terms.

In [1], a function f is defined that maps each formula to a formula that is in normal form, and the following theorem (Theorem 3.2.2 in [1]) is proved.

Theorem 2.13. *If x is a formula, then $\text{EqFSCL} \vdash x = f(x)$.*

As an example, the f -image of $a \wedge \neg b$ is $T \wedge (((a \wedge T) \vee F) \wedge ((\neg b \wedge T) \vee F))$. Note that the segment corresponding to the atom a is $((a \wedge T) \vee F)$, which closely mimics its se-tree, $T \trianglelefteq a \trianglerighteq F$. To further highlight the connection between normal forms and se-trees, we will prove the following corollary:

Corollary 2.14. *If x and y are formulas, then*

$$\text{se}(x) = \text{se}(y) \iff \text{EqFSCL} \vdash x = y \iff \text{EqFSCL} \vdash f(x) = f(y).$$

Proof. The first bi-implication is given by Theorem 2.7. The second follows from the fact that for any E : if $E \vdash x = y$ and $E \vdash y = z$, then $E \vdash x = z$. \square

Thus, this corollary implies that for every formula there is a normal form equivalent that behaves the same, and any other normal form that behaves the same is identified with it by FSCL and all higher logics. Another fundamental property normal forms have is that the three types of normal form (T-term, F-term and T*-term) correspond directly to the three types of trees (closed by T, closed by F, open) seen earlier. This is given by the following proposition:

Proposition 2.15. *Let x be a formula.*

- a. If x is a T-term, then $\text{se}(x)$ is closed by T.*
- b. If x is a F-term, then $\text{se}(x)$ is closed by F.*
- c. If x is a ℓ -term or a T*-term, then $\text{se}(x)$ is open.*

Proof. For (a.), notice that if x and y are formulas and $\text{se}(y)$ is closed by T, then so is $\text{se}(x \vee y)$, as all F's in $\text{se}(x)$ are replaced by $\text{se}(y)$. Since $\text{se}(T) = T$ is closed by T, it follows by a simple inductive proof that se-trees of all T-terms are closed by T. Similarly, for (b.), if $\text{se}(y)$ is closed by F, then so is $\text{se}(x \wedge y)$; this shows that se-trees of F-terms are closed by F. We are left to show (c.).

Suppose x is a T-term and y is a F-term. If we write out $\text{se}((a \wedge x) \vee y)$, we end up with $\text{se}(x) \trianglelefteq a \trianglerighteq \text{se}(y)$. Similarly $\text{se}((\neg a \wedge x) \vee y) = \text{se}(y) \trianglelefteq a \trianglerighteq \text{se}(y)$. Because $\text{se}(x)$ is closed by T and $\text{se}(y)$ is closed by F, the se-tree of a ℓ -term contains both a T leaf and a F leaf. Thus every ℓ -term has an open se-tree.

By Proposition 2.10, the conjunctions and disjunctions added in the P^c and P^d rules keep *-terms open. Finally, if x is a T-term and y a *-term, then $\text{se}(x)$ contains a T leaf and $\text{se}(y)$ is open, so $\text{se}(x \wedge y) = \text{se}(x)[T \mapsto \text{se}(y), F \mapsto F]$ is also open. This means all T*-terms have open se-trees. \square

3. Evaluation and Satisfiability

In propositional logic, the evaluation of a formula depends entirely on which proposition letters are true, and which are not. Once we have assigned a truth value, either true or false, to each proposition letter $p \in \Phi$, the entire formula becomes either true or false. In Short Circuit Logic, the possibility of side-effects somewhat complicates this. Not only can atoms be true or not, but the evaluation of an atom can affect the evaluation of the atoms that come after it. This means that the value assigned to an atom cannot be fixed, but rather depends on what atoms have been evaluated before it. A possible way of defining evaluation for short-circuit logics would be to somehow keep track of the atoms evaluated, and assign a value to an atom based on this ‘evaluation history’. However, as formulas are not bounded in size, such a history-based definition would perhaps be unwieldy.

Instead, we use ‘valuations’ to assign a truth value to each atom. These valuations can be points in a grid, nodes in a graph, etcetera; what they are exactly does not matter, as long as they assign truth values. Side-effects now become transitions between valuations. By moving from one valuation to another, any further atoms are now evaluated in the new valuation, with possibly a different truth values. Thus, a formula can no longer be evaluated as is, but is instead evaluated *at* a certain valuation.

The structures that collect these valuations and the transitions between them, are called ‘valuation algebras’. The definition is based on the definition of valuation algebras for propositional algebra in [2] and the definition of Hoare-McCarthy algebras in [4].

3.1. Valuation Algebras

Definition 3.1. A *valuation algebra* is a non-empty set V , whose elements are called *valuations*, combined with two functions: the *evaluation* $/ : \mathcal{A} \times V \rightarrow \{\text{T}, \text{F}\}$ and the *derivative* $\bullet : \mathcal{A} \times V \rightarrow V$.

So, a valuation algebra is a triple $(V, /, \bullet)$. Instead of the valuations themselves assigning truth values to atoms, we abstract away from what valuations *really* are, and let the function $/$ assign these values for each valuation. The function \bullet describes the transitions between the valuations. We use infix notation for both $/$ and \bullet . Also, if a is an atom, then we speak of ‘the evaluation of a ’ as being the function $a/ : V \rightarrow \{\text{T}, \text{F}\}$, and ‘the derivative of a ’ being $a\bullet : V \rightarrow V$. The reason V must be non-empty is simple: we want to evaluate formulas, and to do so we need at least one valuation.

We often write a valuation algebra simply as V , and use the symbols $/$ and \bullet to implicitly refer to the evaluation and derivative associated with V . We should be cautious

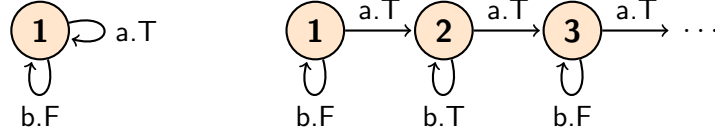


Figure 3.1.: Illustrations for two of the valuation algebras described in Example 3.3.

about this, however. It is worth noting that valuations are just points or worlds or states, that any set of points can be part of a valuation algebras, and that two valuation algebras can have the same set of valuations. What really defines a valuation algebra is its evaluation and derivative. Therefore, if $\mathbf{u} = (V, /, \bullet)$ is a valuation algebra, we shall sometimes emphasise that $/$ and \bullet belong to \mathbf{u} by considering them “in \mathbf{u} ”. We could use subscripts for this, but this would make reading the various equations a bit tiresome.

To be able to evaluate formulas, instead of just atoms, we expand the definition.

Definition 3.2. Let $(V, /, \bullet)$ be a valuation algebra. For each formula x , we define functions $x/H : V \rightarrow \{T, F\}$, the *evaluation* of x , and $x \bullet : V \rightarrow V$, the *derivative* of x , by extending the evaluation a/H and derivative $a \bullet$ for atoms $a \in \mathcal{A}$, as follows:

$$\begin{aligned}
 T/H &= T & T \bullet H &= H \\
 (\neg x)/H &= \neg(x/H) & (\neg x) \bullet H &= x \bullet H \\
 (x \wedge y)/H &= \begin{cases} y/(x \bullet H) & \text{if } x/H = T \\ F & \text{otherwise} \end{cases} & (x \wedge y) \bullet H &= \begin{cases} y \bullet (x \bullet H) & \text{if } x/H = T \\ x \bullet H & \text{otherwise} \end{cases}
 \end{aligned}$$

where x, y are formulas and $H \in V$.

Remark. The following equalities can be derived by for F and \vee :

$$\begin{aligned}
 F/H &= F & F \bullet H &= H \\
 (x \vee y)/H &= \begin{cases} T & \text{if } x/H = T \\ y/(x \bullet H) & \text{otherwise} \end{cases} & (x \vee y) \bullet H &= \begin{cases} x \bullet H & \text{if } x/H = T \\ y \bullet (x \bullet H) & \text{otherwise} \end{cases}
 \end{aligned}$$

where x, y are formulas and $H \in V$.

The definitions concerning T and \neg speak for themselves. In the definition of $(x \wedge y)/$ and $(x \wedge y) \bullet$ the short-circuit nature shows; if x evaluates to false, then $x \wedge y$ immediately evaluates to false as well. The second part, y , is not evaluated and is skipped entirely, thus does not cause any side-effects.

Given a formula x , a valuation algebra V and a specific valuation $H \in V$, we can now evaluate the formula x in H by using these definitions to calculate x/H . The rest of this section will be spent discussing a few properties of valuation algebras.

In Example 3.3, a few valuation algebras are defined for two atoms, a and b . It should be noted that a valuation algebra requires a properly defined evaluation and derivative function for *all* atoms in \mathcal{A} . For practical reasons, we only show two. Figures 3.1 and 3.2 depict the valuation algebras defined in the examples.

Example 3.3. A few examples of valuation algebras.

- a. The valuation algebra $(\{1\}, /, \bullet)$ where $\mathbf{a}/1 = \text{T}$, $\mathbf{b}/1 = \text{F}$, $\mathbf{a} \bullet 1 = 1$ and $\mathbf{b} \bullet 1 = 1$.
- b. The valuation algebra $(\mathbb{N}, /, \bullet)$ where $\mathbf{a}/n = \text{T}$, $\mathbf{b}/n = \text{T}$ if and only if n is odd, $\mathbf{a} \bullet n = n + 1$ and $\mathbf{b} \bullet n = n$ for $n \in \mathbb{N}$.
- c. The valuation algebra $(\mathbb{N}, /, \bullet)$ where $\mathbf{a}/n = \text{T}$ if and only if $n > 1$, $\mathbf{b}/n = \text{T}$ if and only if n is a multiple of 4,

$$\mathbf{a} \bullet n = \begin{cases} n/2 & \text{if } n \text{ is even} \\ n & \text{if } n = 1 \\ 3 \cdot n + 1 & \text{otherwise} \end{cases}$$

and $\mathbf{b} \bullet n = n$ for $n \in \mathbb{N}$.

- d. The valuation algebra $(\mathbb{R}^2, /, \bullet)$ for some fixed sets $A \subseteq \mathbb{R}^2$ and $B \subseteq \mathbb{R}^2$, where $\mathbf{a}/(t_1, t_2)$ if and only if $(t_1, t_2) \in A$ and $\mathbf{b}/(t_1, t_2)$ if and only if $(t_1, t_2) \in B$, and where $\mathbf{a} \bullet (t_1, t_2) = (t_1 + \frac{1}{3}, t_1 + \frac{1}{3})$ and $\mathbf{b} \bullet (t_1, t_2) = (t_1/2, t_2/2)$.

The valuation algebra described in Example 3.3a only has one valuation, which means that there can be no side-effects. Therefore, evaluating a formula in this valuation algebra is similar to evaluating it in propositional logic, i.e. assigning either ‘true’ or ‘false’ to each atom in \mathcal{A} and then resolving the formula. As such, these types of valuation algebras are not very interesting to us.

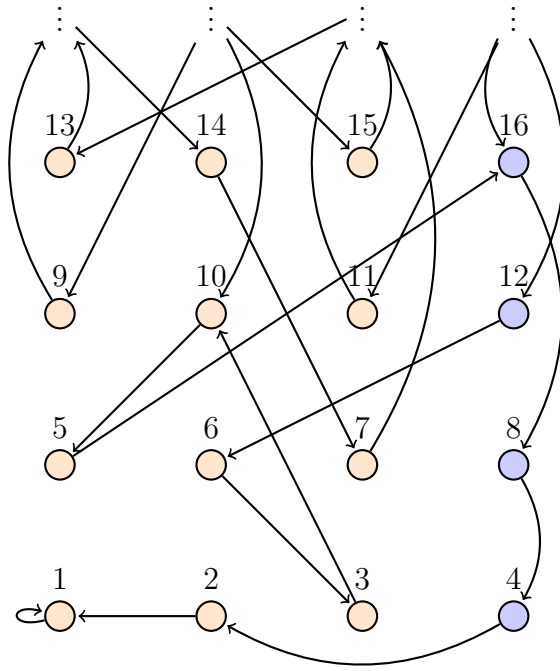
Definition 3.4. A valuation algebra that contains only one valuation is called *trivial*.

The valuation algebra from Example 3.3b is more interesting; it can be thought of as a program fragment based on a positive integer \mathbf{n} , with two functions

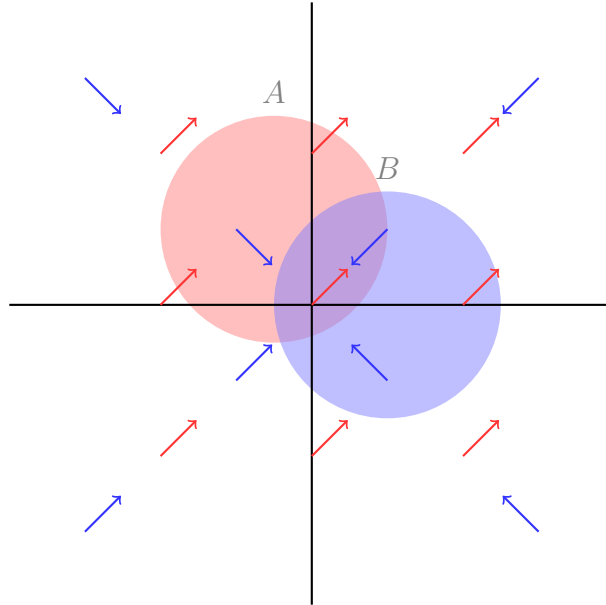
```
boolean a()
{
    n = (n + 1)
    return true
}
boolean b()
{
    return (n % 2 == 0)
}
```

where the C-like $\mathbf{n} \% 2$ returns 0 if \mathbf{n} is even, and 1 if \mathbf{n} is odd.

The third and fourth valuation algebras in Example 3.3 are even more complex. In fact, it is not hard to imagine that there are practically no limits when it comes to ‘inventing’ new valuation algebras, as long as the evaluation and derivative functions are properly defined. The range of valuation algebras is too wild and too expansive to accurately describe in four or five examples. Instead, we will characterise them by their properties.



(a) A segment of the Collatz tree. The arrows represent derivation by **a**. The blue colour indicates where **b** is true.



(b) The sets A and B as a Venn diagram in \mathbb{R}^2 . The arrows indicate the direction of derivation; red for **a**, blue for **b**.

Figure 3.2.: Illustrations for two more valuation algebras described in Example 3.3.

Definition 3.5. Let V be a valuation algebra, then V is called

- ▷ *repetition-proof* if $a/(a \bullet H) = a/H$ for all $a \in \mathcal{A}$ and $H \in V$.
- ▷ *contractive* if V is repetition-proof and $a \bullet a \bullet H = a \bullet H$ for all $a \in \mathcal{A}$ and $H \in V$.
- ▷ *memorizing* if V is contractive and

$$a/(b \bullet a \bullet H) = a/H, \quad a \bullet b \bullet a \bullet H = b \bullet a \bullet H$$

for all $a, b \in \mathcal{A}$ and $H \in V$.

- ▷ *static* if V is memorizing and $a/(b \bullet H) = a/H$ for all $a, b \in \mathcal{A}$ and $H \in V$.

We will denote the collection of all valuation algebras by **fr**, which stands for ‘free’. Moreover, we define **rp**, **cr**, **mem** and **st** as the collections of repetition-proof, contractive, memorizing and static valuation algebras respectively. Note that **st** is a subcollection of **mem**, which is a subcollection of **cr**, etcetera.

From the names alone, one might suspect a link between the five collections of valuation algebras and the five short-circuit logics. The link is this: if two formulas are identified by, say, MSCL, then they ‘behave’ the same under all memorizing valuation algebras. To show this link, we will first need to properly define what it means to behave the same. We will define a relation called ‘valuation congruence’ for each valuation algebra, and we will prove that this relation is in fact a congruence.

Definition 3.6. Let V be a valuation algebra. Two formulas x and y are called *valuation congruent* with respect to V if $x/H = y/H$ and $x \bullet H = y \bullet H$ for all $H \in V$. We denote this by $x \equiv_V y$.

Proposition 3.7. Let V be a valuation algebra, then \equiv_V is a congruence, i.e., for all formulas x, y, x' and y' , if $x \equiv_V x'$ and $y \equiv_V y'$, then $\neg x \equiv_V \neg x'$ and $x \triangleleft y \equiv_V x' \triangleleft y'$.

Proof. Let V be a valuation algebra and let x, y, x', y' be formulas such that $x \equiv_V x'$ and $y \equiv_V y'$. Using the definitions of evaluation and derivative, we find

$$(\neg x)/H = \neg(x/H) = \neg(x'/H) = (\neg x')/H$$

and

$$(\neg x) \bullet H = x \bullet H = x' \bullet H = (\neg x') \bullet H$$

for all $H \in V$. This means $\neg x \equiv_V \neg x'$.

Because $x \bullet H = x' \bullet H$ for all H , we get $y/(x \bullet H) = y/(x' \bullet H)$, and since $y/G = y'/G$ for all G , including $G = x' \bullet H$, we get $y/(x' \bullet H) = y'/(x' \bullet H)$. Also, because $x/H = F \Leftrightarrow x'/H = F$, we find

$$\begin{aligned} (x \triangleleft y)/H &= \begin{cases} y/(x \bullet H) & \text{if } x/H = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} y'/(x' \bullet H) & \text{if } x'/H = T \\ F & \text{otherwise} \end{cases} \\ &= (x' \triangleleft y')/H. \end{aligned}$$

Similarly, because $y \bullet G = y' \bullet G$ for all G , including $G = x' \bullet H$, we get

$$\begin{aligned} (x \wp y) \bullet H &= \begin{cases} y \bullet (x \bullet H) & \text{if } x/H = \text{T} \\ x \bullet H & \text{otherwise} \end{cases} \\ &= \begin{cases} y' \bullet (x' \bullet H) & \text{if } x'/H = \text{T} \\ x' \bullet H & \text{otherwise} \end{cases} \\ &= (x' \wp y') \bullet H \end{aligned}$$

and this proves $x \wp y \equiv_V x' \wp y'$. \square

The following theorem provides the desired connection between the five short-circuit logics and the five collections of valuation algebra. It is not proved in this thesis, but it is based on results proved in [4] and to a lesser extent [1].

Theorem 3.8. *Let x and y be formulas.*

- a. $\text{EqFSCL} \vdash x = y \iff x \equiv_V y$ for all V in **fr**.
- b. $\text{EqRPSCL} \vdash x = y \iff x \equiv_V y$ for all V in **rp**.
- c. $\text{EqCSCL} \vdash x = y \iff x \equiv_V y$ for all V in **cr**.
- d. $\text{EqMSCL} \vdash x = y \iff x \equiv_V y$ for all V in **mem**.
- e. $\text{EqSSCL} \vdash x = y \iff x \equiv_V y$ for all V in **st**.

The properties of memorizing and static valuation algebras are stronger than they may appear at first. This is shown by the following proposition, the proof of which can be found in the appendix.

Proposition 3.9. *Let V be a valuation algebra.*

- a. *If V is memorizing then*

$$x/(y \bullet x \bullet H) = x/H, \quad x \bullet y \bullet x \bullet H = y \bullet x \bullet H$$

for all $H \in V$ and all formulas x, y .

- b. *If V is static then $x/(y \bullet H) = x/H$ for all $H \in V$ and all formulas x, y .*

The property stated in Proposition 3.9b is especially strong. It says that, no matter what formula y we evaluate, its derivative does not alter the evaluation of a formula x . This renders side-effects useless. The following two propositions emphasise this.

Proposition 3.10. *Every trivial valuation algebra is static.*

Proof. It is easy to check that a valuation algebra where $a \bullet H = H$ for all $a \in \mathcal{A}$ and all valuations H , is repetition-proof, contractive, memorizing and static. Clearly all trivial valuation algebras have that property. \square

Proposition 3.11. *Let $(V, /, \bullet)$ be a static valuation algebra and let $H \in V$ be fixed. There exists a trivial valuation algebra $(\{H\}, /_0, \bullet_0)$ such that $x/H = x/_0H$ for every formula x .*

Proof. Let $(V, /, \bullet)$ be a static valuation algebra and let $H \in V$. We construct the valuation algebra $(\{H\}, /_0, \bullet_0)$ by stating $a/_0H = T$ if and only if $a/H = T$, and $a \bullet_0 H = H$ for all $a \in \mathcal{A}$.

We will prove by induction that $x/H = x/_0H$ for all formulas x . The cases T and a for $a \in \mathcal{A}$ are clear. Also, if $x = \neg x_1$ is a formula such that $x_1/H = x_1/_0H$, then $(\neg x_1)/H = \neg(x_1/H) = \neg(x_1/_0H) = (\neg x_1)/_0H$. So suppose $x = x_1 \triangleleft x_2$ such that $x_i/H = x_i/_0H$. Then we use Proposition 3.9 to get

$$\begin{aligned} x/H &= \begin{cases} x_2/(x_1 \bullet H) & \text{if } x_1/H = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2/H & \text{if } x_1/H = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2/_0H & \text{if } x_1/_0H = T \\ F & \text{otherwise} \end{cases} \\ &= x/_0H. \end{aligned}$$

This concludes the proof. \square

This shows that for any valuation H in a static valuation algebra, evaluating a formula in H is essentially the same as evaluating it in a propositional logic sense. However, this proposition does not imply that all static valuation algebras are somehow ‘equivalent’ to trivial valuation algebras. One might imagine a static valuation algebra consisting of multiple valuations, but without any ‘transitions’ between the valuations. A formula evaluated in different valuations of such a valuation algebra could have different outcomes, which is impossible in a trivial valuation algebra.

Still, these two propositions show that SSCL is arguably the least interesting short-circuit logic in terms of evaluation and satisfiability.

3.2. Satisfiability

Now that we have defined what it means to evaluate a formula, we can define what it means for a formula to be satisfiable.

Definition 3.12. Let K be a collection of valuation algebras. A formula x is *satisfiable* with respect to K if there exists a V in K such that $x/H = T$ for some $H \in V$, and we denote this by $\mathbf{SAT}_K(x)$. A formula x is *falsifiable* w.r.t. K if there exists a V in K such that $x/H = F$ for some $H \in V$, and we denote this by $\mathbf{FAL}_K(x)$.

Thus, to show that a formula is satisfiable, it is enough to find or construct a valuation algebra that ‘satisfies’ the formula. Conversely, to show that a formula is not satisfiable, we need to prove that for every valuation in every valuation algebra within a certain

collection, the formula evaluates to F. It is not enough to show that the formula is falsifiable; in fact, most formulas will be both satisfiable and falsifiable. Also, note that $\mathbf{FAL}_K(x) \Leftrightarrow \mathbf{SAT}_K(\neg x)$. This means that for every formula x , at least one of $\mathbf{SAT}_K(x)$ and $\mathbf{FAL}_K(x)$ must be true.

Also, if we already have a valuation algebra that satisfies a formula, then it may be part of multiple collections and therefore prove multiple types of satisfiability. In particular, the collections **fr**, **rp**, **cr**, **mem** and **st** are related, so we immediately find the following proposition.

Proposition 3.13. *Let x be a formula, then*

$$\mathbf{SAT}_{\mathbf{st}}(x) \Rightarrow \mathbf{SAT}_{\mathbf{mem}}(x) \Rightarrow \mathbf{SAT}_{\mathbf{cr}}(x) \Rightarrow \mathbf{SAT}_{\mathbf{rp}}(x) \Rightarrow \mathbf{SAT}_{\mathbf{fr}}(x).$$

Proof. This follows directly from the definition. \square

The following theorem further strengthens the connection between the five short-circuit logics and our definition of satisfiability.

Proposition 3.14. *Let K be a collection of valuation algebras, and let x and y be formulas. If $x \equiv_V y$ for all V in K , then $\mathbf{SAT}_K(x) \Leftrightarrow \mathbf{SAT}_K(y)$.*

Proof. Let K be a collection of valuation algebras, and let x and y be formulas such that $x \equiv_V y$ for all V in K . If $\mathbf{SAT}_K(x)$, then there exists a V_0 in K such that $x/H_0 = \mathbf{T}$ for some $H_0 \in V_0$. Because $x \equiv_{V_0} y$, we find $y/H_0 = \mathbf{T}$, and thus $\mathbf{SAT}_K(y)$. If $\neg \mathbf{SAT}_K(x)$, then for every V in K , it must be that $x/H = \mathbf{F}$ for all $H \in V$. But for every V in K we have $x \equiv_V y$, thus $y/H = \mathbf{F}$ for all $H \in V$. This shows $\neg \mathbf{SAT}_K(y)$. \square

Theorem 3.15. *Let x and y be formulas.*

- a. *If $\text{EqFSCL} \vdash x = y$, then $\mathbf{SAT}_{\mathbf{fr}}(x) \Leftrightarrow \mathbf{SAT}_{\mathbf{fr}}(y)$.*
- b. *If $\text{EqRPSCL} \vdash x = y$, then $\mathbf{SAT}_{\mathbf{rp}}(x) \Leftrightarrow \mathbf{SAT}_{\mathbf{rp}}(y)$.*
- c. *If $\text{EqCSCL} \vdash x = y$, then $\mathbf{SAT}_{\mathbf{cr}}(x) \Leftrightarrow \mathbf{SAT}_{\mathbf{cr}}(y)$.*
- d. *If $\text{EqMSCL} \vdash x = y$, then $\mathbf{SAT}_{\mathbf{mem}}(x) \Leftrightarrow \mathbf{SAT}_{\mathbf{mem}}(y)$.*
- e. *If $\text{EqSSCL} \vdash x = y$, then $\mathbf{SAT}_{\mathbf{st}}(x) \Leftrightarrow \mathbf{SAT}_{\mathbf{st}}(y)$.*

Proof. This follows by combining Theorem 3.8 and Proposition 3.14. \square

Lastly, the following corollary reinforces the idea that SSCL and propositional logic are very similar, especially regarding satisfiability.

Corollary 3.16. *Let x be a formula. Then $\mathbf{SAT}_{\mathbf{st}}(x)$ if and only if one can assign either ‘true’ or ‘false’ to each $a \in \mathcal{A}$ such that x , as a propositional formula, is true.*

Proof. This follows from Proposition 3.10 and Proposition 3.11. \square

4. Path-Satisfiability

The definitions of evaluation and satisfiability discussed in the previous chapter match the theoretical desires we have for them. However, implementing them seems impossible, or at least highly impractical. They allow all kinds of valuations, which is good, but this generic and abstract nature does not fit the finite and discrete world of a computer program. We therefore need to define an alternative form of evaluation.

As we have already seen how evaluation trees emulate the short-circuit behaviour of our formulas, we will use them as a basis. The basic idea is that a formula can be made true if there is a route, or a ‘path’, through its se-tree to a T leaf. We will formalise this by defining ‘valuation paths’ and their result on trees.

4.1. Valuation Paths

Definition 4.1. A *valuation path* of length n is a sequence $\langle p_1, \dots, p_n \rangle$, where each p_i is a pair $(u_i, b_i) \in \mathcal{A} \times \{T, F\}$.

Each of the segments of a valuation path consists of an atom from \mathcal{A} and a truth value that states whether this atom should be true or not. There is one valuation path of length 0, which we will call ϵ . If P is a valuation path of length n , we write $|P| = n$. To effectively work with valuation paths, we need to be able to manipulate them by adding other valuation paths to them.

Definition 4.2. Let $P = \langle p_1, \dots, p_n \rangle$ and $Q = \langle q_1, \dots, q_m \rangle$ be two valuation paths of length n and m respectively. The *concatenation* of P and Q is the valuation path $P \cdot Q := \langle p_1, \dots, p_n, q_1, \dots, q_m \rangle$ of length $n + m$.

Note that concatenating ϵ to a valuation path has no effect, that is, $\epsilon \cdot P = P = P \cdot \epsilon$. We will also want to use induction and recursion on valuation paths; to this end, note that every valuation path P of positive length can be made by concatenating its first segment with the rest of the valuation path. Thus $P = (u, b) \cdot Q$ for some $u \in \mathcal{A}$, some $b \in \{T, F\}$ and some valuation path Q with $|P| = |Q| + 1$.

Using this, we can now define a valuation path’s ‘result’ on a tree. If we apply a valuation path starting with an atom $u \in \mathcal{A}$ to a tree with the same atom u as its root, then the truth value b associated with it determines whether we proceed with the left or the right branch. We iterate this process, until we reach a leaf. If it is a T leaf, the result is T, and if it is a F leaf, the result is F. However, we must also consider the cases where the valuation path and the tree do not match up. In these cases, we leave the result undefined. Figure 4.1 shows this. Formally:

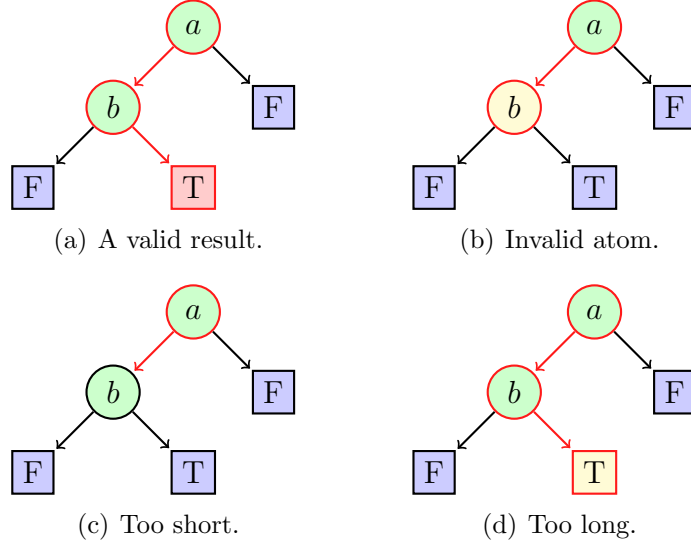


Figure 4.1.: The result of the valuation path $\langle (a, T), (b, F) \rangle$ in the tree $se(a \wedge \neg b)$ is T. The results of $\langle (a, T), (a, F) \rangle$, $\langle (a, T) \rangle$ and $\langle (a, T), (b, F), (a, F) \rangle$ in that same tree are all undefined.

Definition 4.3. The *result* of a valuation path P on a tree X , denoted $P : X$, is either an element of $\{T, F\}$ or undefined. We define $P : X$ recursively, as follows:

$$\begin{aligned}
 \epsilon : T &= T \\
 \epsilon : F &= F \\
 ((u, b) \cdot Q) : (X_1 \leq a \geq X_2) &= \begin{cases} Q : X_1 & \text{if } u = a \text{ and } b = T \\ Q : X_2 & \text{if } u = a \text{ and } b = F \end{cases}
 \end{aligned}$$

and for all other circumstances, we leave $P : X$ undefined.

Eventually, we want to relate this back to formulas, as generic trees are not the most interesting to us. In the rest of this section, we will discuss what results valuation paths have on se-trees.

First, consider the following: we have two trees, X and Y , and a path P . If P results to either T of F on X , then that means that P leads us through X to one of the leaves of X . If where to replace this leaf with Y , then P would lead us to the root of Y . Intuitively, we want to be able to continue the path where we left of and traverse Y as well, by appending another path to P . The following proposition allows us to do so.

Proposition 4.4. *Let X, Y, Y' be trees and P, Q paths. If $P : X$ is defined, then $(P \cdot Q) : X[\beta \mapsto Y, \neg\beta \mapsto Y'] = Q : Y$ where $\beta = P : X$.*

Proof. Let Y and Y' be trees and Q a path. We prove this proposition by induction to the depth of X . Call X “compatible” if, for all paths P ,

$$\text{if } \beta = P : X \text{ is defined, then } (P \cdot Q) : X[\beta \mapsto Y, \neg\beta \mapsto Y'] = Q : Y.$$

The only trees of depth 0 are T and F. Let X be either. If P is a path such that $P : X$ is defined, then $P = \epsilon$, thus $P \cdot Q = Q$. If $X = T$ then $P : X = T$, which means $Z = T[T \mapsto Y, F \mapsto Y']$; if not, then $X = F$, $P : X = F$ and $Z = F[T \mapsto Y', F \mapsto Y]$. In either case, $Z = Y$, thus $(P \cdot Q) : Z = Q : Z = Q : Y$. We conclude that all trees of depth 0 are compatible.

Let $n \geq 0$ and assume that all trees of depth at most n are compatible. Let X be a tree of depth $n + 1$, then $X = X_1 \trianglelefteq a \trianglerighteq X_2$ for trees X_1 and X_2 and for some $a \in \mathcal{A}$. Then X_1 and X_2 are of depth at most n , thus compatible. To complete the proof, we need to show that X is compatible.

Let P be a path such that $\beta = P : X$ is defined, then P must be of the form $P = (a, b) \cdot R$ for some $b \in \{T, F\}$ and some path R . We get

$$\beta = P : X = ((a, b) \cdot R) : (X_1 \trianglelefteq a \trianglerighteq X_2) = \begin{cases} R : X_1 & \text{if } b = T \\ R : X_2 & \text{if } b = F \end{cases}$$

and this means that if $b = T$ then $R : X_1 = \beta$, and if $b = F$ then $R : X_2 = \beta$. Let $Z = X[\beta \mapsto Y, \neg\beta \mapsto Y']$, then $Z = Z_1 \trianglelefteq a \trianglerighteq Z_2$ where $Z_i = X_i[\beta \mapsto Y, \neg\beta \mapsto Y']$. Because X_1 and X_2 are compatible, we find

$$\begin{aligned} (P \cdot Q) : Z &= ((a, b) \cdot (R \cdot Q)) : (Z_1 \trianglelefteq a \trianglerighteq Z_2) \\ &= \begin{cases} (R \cdot Q) : Z_1 & \text{if } b = T \\ (R \cdot Q) : Z_2 & \text{if } b = F \end{cases} \\ &= \begin{cases} Q : Y & \text{if } b = T \\ Q : Y & \text{if } b = F \end{cases} \\ &= Q : Y. \end{aligned}$$

Therefore X is compatible. □

The next proposition allows us to say something useful about the results of valuation paths on se-trees; namely that they are what we might expect them to be.

Proposition 4.5. *Let x, y be formulas and P, Q paths. If $P : \text{se}(x)$ is defined, then*

$$\begin{aligned} P : \text{se}(\neg x) &= \neg(P : \text{se}(x)) \\ (P \cdot Q) : \text{se}(x \wp y) &= \begin{cases} Q : \text{se}(y) & \text{if } P : \text{se}(x) = T \\ Q : F & \text{otherwise} \end{cases} \end{aligned}$$

Proof. Let x be a formula, let $X = \text{se}(x)$ and let P a path such that $P : X$ is defined. Let $Q = \epsilon$ so that $P \cdot Q = P$. If $P : X = T$, then let $Y = F$ and $Y' = T$ which gives us $\text{se}(\neg x) = X[T \mapsto Y, F \mapsto Y']$. Now we can use Proposition 4.4 in order to get $P : \text{se}(\neg x) = Q : Y = \epsilon : F = F$. Otherwise let $Y = T$ and $Y' = F$, which gives $\text{se}(\neg x) = X[T \mapsto Y', F \mapsto Y]$. By the proposition, $P : \text{se}(\neg x) = Q : Y = \epsilon : T = T$. Either way, we find $\neg(P : \text{se}(x))$.

Let x, y be formulas, $X = \text{se}(x)$ and let P, Q paths such that $P : X$ is defined. If $P : X = T$, then let $Y = \text{se}(y)$ and $Y' = F$, thus $\text{se}(x \wp y) = X[T \mapsto Y, F \mapsto Y']$. The

proposition tells us $(P \cdot Q) : \text{se}(x \triangleleft y) = Q : \text{se}(y)$. Otherwise, let $Y = F$ and $Y' = \text{se}(y)$, and thus $\text{se}(x \triangleleft y) = X[T \mapsto Y', F \mapsto Y]$. Thus $(P \cdot Q) : \text{se}(x \triangleleft y) = Q : F$ by the proposition. \square

We also want a converse to the previous proposition. That is, if a path traverses a ‘compound’ tree to a leaf of that tree, then some initial part of this path will lead us to the point where the substitution took place. More formally:

Proposition 4.6. *Let X, Y, Y' be trees and P a path. If $P : X[T \mapsto Y, F \mapsto Y']$ is defined, then there are paths R and Q with $P = R \cdot Q$, such that $R : X$ is defined and*

$$P : X[T \mapsto Y, F \mapsto Y'] = \begin{cases} Q : Y & \text{if } R : X = T \\ Q : Y' & \text{otherwise} \end{cases} \quad (\star)$$

Proof. Let Y and Y' be trees. We also prove this proposition by induction, but this time to the length of P . Call P “divisible” if for every tree X there are R, Q such that $P = R \cdot Q$ and

$$\text{if } P : X[T \mapsto Y, F \mapsto Y'] \text{ is defined, then } R : X \text{ is defined and } (\star).$$

The only path of length P is ϵ . Let X be a tree and let $Z = X[T \mapsto Y, F \mapsto Y']$. Let $R = \epsilon$ and $Q = \epsilon$. If $\epsilon : Z$ is defined, then $Z \in \{T, F\}$, thus $X, Y, Y' \in \{T, F\}$. If $X = T$, then $R : X = T$ and $Z = Y$. If $X = F$, then $R : X = F$ and $Z = Y'$. Either way, (\star) holds and ϵ is divisible.

Let $n \geq 0$ and assume all paths of length at most n are divisible. Let P be of length $n + 1$. To complete the proof, we need to show that P is divisible. Note that if $X \in \{T, F\}$ and $Z = X[T \mapsto Y, F \mapsto Y']$, then either $Z = Y$ or $Z = Y'$ and we can take $R = \epsilon$ and $Q = P$ to immediately get the result. Thus in the following we assume that $X = X_1 \triangleleft a \triangleright X_2$ for some trees X_1, X_2 and some $a \in \mathcal{A}$, and this gives us $Z = Z_1 \triangleleft a \triangleright Z_2$ where $Z_i = X_i[T \mapsto Y, F \mapsto Y']$.

Because $P \neq \epsilon$, we can write $P = (u, b) \cdot P'$ for some $u \in \mathcal{A}$, some $b \in \{T, F\}$ and some path P' of length n . Suppose $P : Z$ is defined, then $u = a$ and we get

$$P : Z = ((a, b) \cdot P') : (Z_1 \triangleleft a \triangleright Z_2) = \begin{cases} P' : Z_1 & \text{if } b = T \\ P' : Z_2 & \text{otherwise} \end{cases}$$

thus $P' : Z_i = P : Z$ is defined, where $i = 1$ if $b = T$ and $i = 2$ otherwise.

Because P' is divisible, there are R', Q' such that $P' = R' \cdot Q'$, that $R' : X_i$ is defined and

$$P' : Z_i = \begin{cases} Q' : Y & \text{if } R' : X_i = T \\ Q' : Y' & \text{otherwise} \end{cases}$$

Take $R = (u, b) \cdot R'$ and $Q = Q'$, then $P = (u, b) \cdot P' = ((u, b) \cdot R') \cdot Q'$. We find that $R : X = R' : X_i$ is defined by our choice of i , and (\star) follows. Thus P is divisible. \square

4.2. Path-Satisfiability

In the previous section, we have defined an alternative way to evaluate formulas, based on their se-tree. Using this, we can now define our alternative satisfiability, called ‘path-satisfiability’. In principle, a formula is path-satisfiable if there is a path that results in T on the formula’s se-tree, and path-falsifiable if there is a path that results in F.

However, this definition alone gives us no method allow or disallow certain side-effects, which we need to correspond to our five short-circuit logics. To this purpose we define two properties for valuation paths: ‘repetition-proof’ and ‘memorizing’.

Definition 4.7. Let $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ be a valuation path, then P is called

▷ *repetition-proof* if $u_i = u_{i+1} \implies b_i = b_{i+1}$ for all $i < n$.

▷ *memorizing* if $u_i = u_j \implies b_i = b_j$ for all $i, j \leq n$.

Of course, every memorizing valuation path is also repetition-proof. Now we can formally define three forms of path-satisfiability; one ‘free’ path-satisfiability that is without any requirements, and one path-satisfiability for each of the two properties defined above.

Definition 4.8. Let x be a formula. A formula is *path-satisfiable* if there exists a valuation path P such that $P : \text{se}(x) = \text{T}$, and we denote this $\mathbf{PathSat}_{fr}(x)$. A formula is *rp-path-satisfiable*, denoted $\mathbf{PathSat}_{rp}(x)$, if there is a repetition-proof path, and *mem-path-satisfiable*, denoted $\mathbf{PathSat}_{mem}(x)$, if there is a memorizing path.

We also define three analogous forms of path-falsifiability, where $P : \text{se}(x) = \text{F}$, and we denote these by $\mathbf{PathFal}_{fr}(x)$, $\mathbf{PathFal}_{rp}(x)$ and $\mathbf{PathFal}_{mem}(x)$.

If a tree has no T leaves, then there clearly cannot be a valuation path that results in T on this tree. It is not hard to see that if all kinds of valuation path are allowed, the converse is also true; if a tree has a T leaf, then there is a valuation path that results in T on this tree. This is stated by the following proposition.

Proposition 4.9. *Let x be a formula.*

a. $\mathbf{PathSat}_{fr}(x)$ if and only if $\text{se}(x)$ has a T leaf.

b. $\mathbf{PathFal}_{fr}(x)$ if and only if $\text{se}(x)$ has a F leaf.

Proof. If a tree of the form $X \trianglelefteq a \trianglerighteq Y$ contains a leaf, then this leaf can be reached by a valuation path either of the form $(a, \text{T}) \cdot P$ where P runs through X , or of the form $(a, \text{F}) \cdot Q$ where Q runs through Y . From this, both statements follow. \square

This proposition has two corollaries that relate to constant-free formulas and formulas in normal form.

Corollary 4.10. *If x is constant-free formula, then $\mathbf{PathSat}_{fr}(x)$ and $\mathbf{PathFal}_{fr}(x)$.*

Proof. This follows from Corollary 2.11 and Proposition 4.9. \square

Corollary 4.11. *Let x be a formula.*

- a. $\mathbf{PathSat}_{fr}(x)$ and $\neg\mathbf{PathFal}_{fr}(x)$ if and only if $f(x)$ is a T-term.
- b. $\neg\mathbf{PathSat}_{fr}(x)$ and $\mathbf{PathFal}_{fr}(x)$ if and only if $f(x)$ is a F-term.
- c. $\mathbf{PathSat}_{fr}(x)$ and $\mathbf{PathFal}_{fr}(x)$ if and only if $f(x)$ is a T*-term.

Proof. This follows by combining Proposition 2.15 and Proposition 4.9. \square

For repetition-proof and memorizing paths, a weaker version of this last corollary exists.

Corollary 4.12. *Let x be a formula.*

- a. If $f(x)$ is a T-term, then $\mathbf{PathSat}_{mem}(x)$ and $\neg\mathbf{PathFal}_{mem}(x)$.
- b. If $f(x)$ is a F-term, then $\neg\mathbf{PathSat}_{mem}(x)$ and $\mathbf{PathFal}_{mem}(x)$.

Proof. We can certainly construct a memorizing valuation path P such that $P : se(x)$ is defined, for example by simply assigning T to all atoms. By Proposition 2.15, if $f(x)$ is a T-term then $se(x)$ is closed by T. This means $P : se(x) = T$. And of course, if $se(x)$ has no F-leaves, then no valuation path Q exists with $Q : se(x) = F$. Analogous statements can be made when $f(x)$ is a F-term. \square

These three corollaries may suggest that path-satisfiability is somewhat trivial to solve. However, most formulas will not be constant-free, and in Chapter 5 we will discuss how normal forms are not ideal to solve path-satisfiability.

Before we continue, an analogue to Proposition 3.13.

Proposition 4.13. *Let x be a formula, then*

$$\mathbf{PathSat}_{mem}(x) \implies \mathbf{PathSat}_{rp}(x) \implies \mathbf{PathSat}_{fr}(x)$$

Proof. This follows directly from the definitions. \square

In Chapter 5 we will discuss an implementation of path-satisfiability. However, our original goal was to describe and implement “real” satisfiability. If our two forms of evaluation and satisfiability do not match up, we have effectively wasted our time defining and proving something unrelated. Figure 4.2 illustrates this disconnect. As we will prove the connections between the types of satisfiability, we will update this illustration.

To show a first connection between valuation algebras and valuation paths, consider the following: suppose we have a formula x that we are evaluating in some valuation algebra, and suppose we make a note each time we encounter an atom, both of which atom it is and of what truth value it is assigned. Then at the end we have a ‘diary’ of sorts, and this diary is in fact a valuation path. The following definition formalises this procedure.



Figure 4.2.: A schematic overview of satisfiability and path-satisfiability. The five green nodes on the left represent satisfiability, as described in Section 3.2 for the five logics described in Section 2.3. The descending dashed arrows between them are given by Proposition 3.13. The three red nodes on the right represent the three types of path-satisfiability defined in Section 4.2, and the descending arrows between them are given by Proposition 4.13.

Definition 4.14. Let V be a valuation algebra. For a formula x and a valuation $H \in V$, we define the *evaluation path* of x at H , denoted by $x \diamond H$, as follows:

$$\begin{aligned} \mathsf{T} \diamond H &= \epsilon \\ a \diamond H &= \langle (a, a/H) \rangle \\ (\neg x) \diamond H &= x \diamond H \\ (x \wp y) \diamond H &= \begin{cases} (x \diamond H) \cdot (y \diamond (x \bullet H)) & \text{if } x/H = \mathsf{T} \\ x \diamond H & \text{otherwise} \end{cases} \end{aligned}$$

The name ‘evaluation path’ refers to how this valuation path is created while evaluating the formula. The purpose of these evaluation paths is that the result of an evaluation path on the se-tree of a formula is exactly the same as the evaluation of the formula in the valuation. Proposition 4.16 states this useful fact.

Proposition 4.15. *Let V be a valuation algebra. For a formula x and some $H \in V$, let $x \diamond H = \langle p_1, \dots, p_n \rangle$ with $p_i = (u_i, b_i)$. Then $b_i = u_i / (u_{i-1} \bullet \dots \bullet u_1 \bullet H)$ for $1 \leq i \leq n$.*

Proof. This is easy to check using Proposition 4.5 and Proposition 4.6. \square

Proposition 4.16. *Let V be a valuation algebra. Then $(x \diamond H) : \text{se}(x) = x/H$ for every formula x and every $H \in V$.*

Proof. This is easy to check using Proposition 4.15. \square

This means that for every formula, if there is a valuation algebra where the formula evaluates to T , then there is also a valuation path whose result in the formula’s se-tree is T . In fact, this valuation path will have the similar properties to the valuation algebra.

Proposition 4.17. *Let V be a valuation algebra, let x be a formula and let $H \in V$.*

- a. If V is repetition-proof, then $x \diamond H$ is repetition-proof.*
- b. If V is memorizing, then $x \diamond H$ is memorizing.*

Proof. This is easy to check using Proposition 4.15. \square

We can now state the following result, which establish one half of the connection between satisfiability and path-satisfiability that we are trying to prove.

Theorem 4.18. *Let x be a formula.*

- a. If $\mathsf{SAT}_{\mathbf{fr}}(x)$, then $\mathsf{PathSat}_{\mathbf{fr}}(x)$.*
- b. If $\mathsf{SAT}_{\mathbf{rp}}(x)$, then $\mathsf{PathSat}_{\mathbf{rp}}(x)$.*
- c. If $\mathsf{SAT}_{\mathbf{mem}}(x)$, then $\mathsf{PathSat}_{\mathbf{mem}}(x)$.*

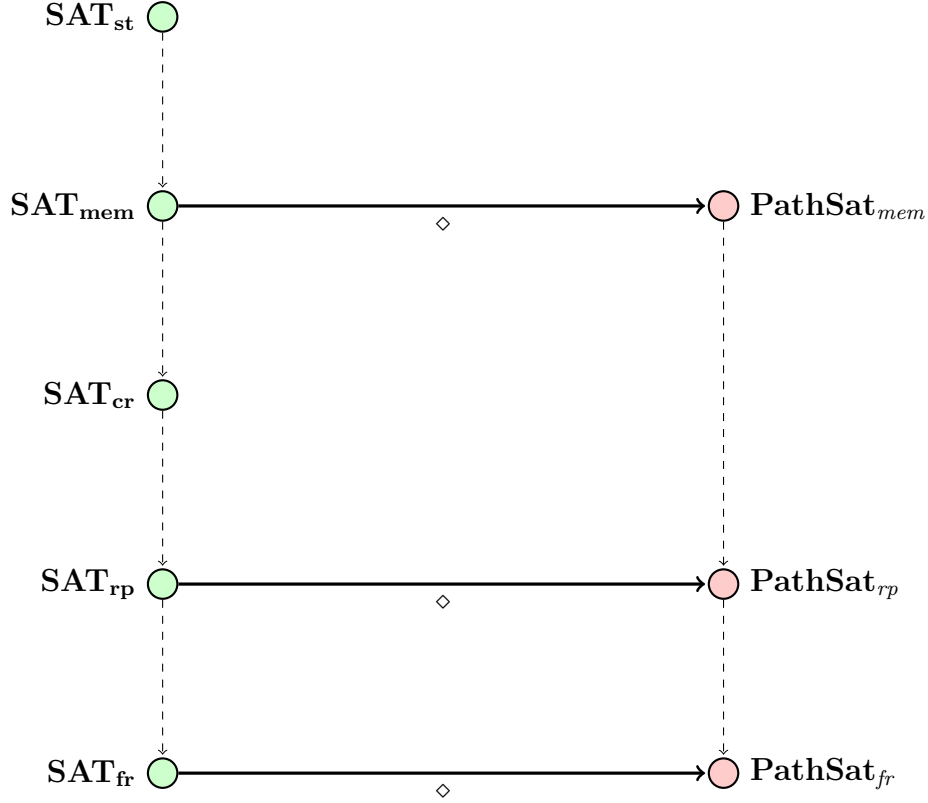


Figure 4.3.: An updated overview, based on Figure 4.2. The three thick arrows labeled \diamond are given by Theorem 4.18.

Proof. Let x be a formula such that $\mathbf{SAT}_{\text{fr}}(x)$, and let V be a valuation algebra with $H \in V$ such that $x/H = \text{T}$. Let $P = x \diamond H$. By Proposition 4.16, we have $P : \text{se}(x) = \text{T}$. This means $\mathbf{PathSat}_{\text{fr}}(x)$.

If $\mathbf{SAT}_{\text{rp}}(x)$ (resp. $\mathbf{SAT}_{\text{mem}}(x)$), then we can find V so that additionally V is repetition-proof (resp. memorizing). By Proposition 4.17, P is repetition-proof (resp. memorizing), which means $\mathbf{PathSat}_{\text{rp}}(x)$ (resp. $\mathbf{PathSat}_{\text{mem}}(x)$). \square

Now we have shown an important connection. Figure 4.3 illustrates this. The next two sections will be spent establishing a converse connection.

4.3. Norm-based Constructors

To show a connection between path-satisfiability and satisfiability, we need to solve the following problem: suppose we have found a valuation path that results in T on the se-tree of a given formula; how do we create a valuation algebra where the formula evaluates to T? At first glance, this seems relatively easy since we can add as many valuations as we need, and each valuation can assign whichever truth value we want to each atom. For each atom we come across, we make a valuation that makes this

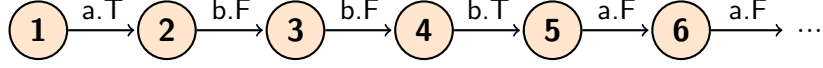


Figure 4.4.: A first attempt to create a valuation algebra $(\mathbb{N}, /, \bullet)$ for the valuation path $P = \langle (a, T), (b, F), (b, F), (b, T), (a, F), (a, F) \rangle$.

atom true and then we jump to the next valuation for the next atom. Thus, for a path $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ we might make a valuation algebra $(\mathbb{N}, /, \bullet)$ such that $a/i = b_i$ and $a \bullet i = i + 1$ for all i , as depicted in Figure 4.4.

Such a valuation algebra could work if the remaining gaps in its definition are filled; however, it has proven difficult to properly write down and prove the propositions that we would need to use such a valuation algebra. We would much rather use a recursive definition, which would allow us to prove our proofs using induction. Therefore, we will only construct finite valuation algebras, and their size will depend on the “size” of the valuation path. We might need different ways to assign a size to a valuation path, and this is achieved by defining norms.

Definition 4.19. A *norm* on valuation paths is a function $\|\cdot\|$ that maps a valuation path P to a value $\|P\| \geq 0$ such that $\|\epsilon\| = 0$ and $\|P \cdot Q\| \leq \|P\| + \|Q\|$.

One norm was already defined in Section 4.1: the length norm $|\cdot|$. Note that, for paths P and Q , $|P \cdot Q| = |P| + |Q|$ and that if $|P| = 0$ then $P = \epsilon$. Traditionally this last property is an additional condition of norms, and functions without it are called “semi-norms”; however, we ignore this distinction. Therefore, the trivial norm defined by $\|P\| = 0$ for all paths P is also a norm.

In this section we will define a few ‘constructors’ that assign a valuation algebra to each valuation path. To effectively use recursion and induction, we need that if a valuation path P is a concatenation of Q and R , then the valuation algebra associated with P should somehow resemble a combination of the two valuation algebras associated to Q and R . However, we do not have a way to combine valuation algebras. Instead, we will try to create constructors that are ‘invariant’ to concatenation; that is, the valuation algebra of a path P is ‘embedded’ in the valuation algebra of any path of the form $R_1 \cdot P \cdot R_2$. These vague notions will be properly defined later in this section. First, we define what kind of constructors we will make.

Definition 4.20. Let $\|\cdot\|$ be a norm. If for each valuation path P a valuation algebra $\mathbf{u}(P)$ of the form $(\{1, \dots, \|P\| + 1\}, /, \bullet)$ is defined such that $i \leq (a \bullet i) \leq i + 1$ for all i and all $a \in \mathcal{A}$, then \mathbf{u} is a *norm-based constructor* for $\|\cdot\|$.

As desired, if \mathbf{u} is a norm-based constructor then the size of $\mathbf{u}(P)$ depends linearly on $\|P\|$. Note that the second property states that for each valuation i and each $a \in \mathcal{A}$, either a does not change i or it advances i by one; it cannot ‘jump’ forward and it cannot go back. This will help us in making these constructors ‘invariant’.

We are now ready to create our first norm-based constructor, called *va*.

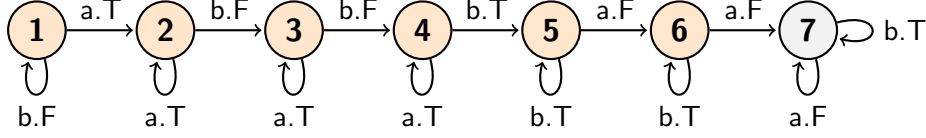


Figure 4.5.: An illustration of the valuation algebra $\text{va}(P)$, again for the valuation path $P = \langle (a, T), (b, F), (b, F), (b, T), (a, F), (a, F) \rangle$.

Definition 4.21. Let $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ be a valuation path. For $a \in \mathcal{A}$ and $k \leq n + 1$, we define $\text{last}(a, k)$ as the largest $i \leq n$ such that $i \leq k$ and $u_i = a$, or 0 if no such i exists. We define $\text{va}(P)$ as the valuation algebra $(\{1, \dots, n + 1\}, /, \bullet)$, where $/$ and \bullet are defined by

$$a/i = \begin{cases} b_j & \text{if } j = \text{last}(a, i) > 0 \\ \text{F} & \text{otherwise} \end{cases}$$

$$a \bullet i = \begin{cases} i + 1 & \text{if } i \leq n \text{ and } u_i = a \\ i & \text{otherwise} \end{cases}$$

for $a \in \mathcal{A}$ and $i \leq n + 1$.

Note that va is a norm-based constructor for the length norm $|\cdot|$. Similar to our earlier idea, the valuation algebra $\text{va}(P)$ for a path $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ has the desirable properties that $u_i/i = b_i$ and $u_i \bullet i = i + 1$, but this time for a finite amount of valuations instead of for all \mathbb{N} . As a counterpart to Figure 4.4, the valuation algebra $\text{va}(P)$ is depicted in Figure 4.5 for the same valuation path P .

From our illustrated example, it is clear that $\text{va}(P)$ shares some features with P . In Section 4.4, we will prove a very strong result about the norm-based constructor va :

Lemma (4.28a). *Let x be a formula and let P a valuation path such that $P : \text{se}(x)$ is defined. Then $x/1 = P : \text{se}(x)$ in $\text{va}(P)$.*

In particular, if $P : \text{se}(x) = \text{T}$, then $x/1 = \text{T}$ in $\text{va}(P)$. As a consequence, each path-satisfiable formula is satisfiable with respect to **fr**, and with the use of the following proposition, each rp-path-satisfiable formula is satisfiable with respect to **rp**.

Proposition 4.22. *If P is a repetition-proof valuation path, then $\text{va}(P)$ is a repetition-proof valuation algebra.*

Proof. Let P be a repetition-proof valuation path of length n . Let $i \leq n + 1$ and $a \in \mathcal{A}$, then we need to show that $a/(a \bullet i) = a/i$ in $\text{va}(P)$. If $a \bullet i = i$, then this is clear, so we can suppose that $i \leq n$ and $a \bullet i = i + 1$. This means $u_i = a$ and $a/i = b_i$. Now consider $a/(i + 1)$; clearly $i \leq \text{last}(a, i + 1) \leq i + 1$, but this means that either $a/(i + 1) = b_i$ or $u_{i+1} = a$ and $a/(i + 1) = b_{i+1}$. In the latter case, $b_{i+1} = b_i$ follows as P is repetition-proof. \square

This sounds like a great result, and we can expand on Figure 4.3. However, some care must be taken here. If we were to only show a connection from $\mathbf{PathSat}_{rp}$ to \mathbf{SAT}_{rp} , and one from $\mathbf{PathSat}_{mem}$ to \mathbf{SAT}_{mem} , then we would leave \mathbf{SAT}_{cr} and \mathbf{SAT}_{st} without path-related equivalents. This would imply that our three path-satisfiabilities are insufficient to describe the five different satisfiabilities. Instead, we will show connections from $\mathbf{PathSat}_{rp}$ directly to \mathbf{SAT}_{cr} and similarly from $\mathbf{PathSat}_{mem}$ to \mathbf{SAT}_{st} . The implications of this will be discussed in Chapter 6; for now, we are concerned with constructing appropriate valuation algebras.

Unfortunately, our example in Figure 4.5 suggests that the valuation algebras created by va will not be contractive for most repetition-proof valuation paths, so va will not do. The problem lies in the following: if a valuation path P has two subsequent segments where the atoms are the same, i.e. $\langle (u_1, b_1), \dots, (u_i, b_i), (u_{i+1}, b_{i+1}), \dots, (u_n, b_n) \rangle$ with $u_i = u_{i+1}$, then $va(P)$ is not contractive. On the other hand, any P where $u_i \neq u_{i+1}$ is clearly repetition-proof, and $va(P)$ will be contractive.

Proposition 4.23. *Let $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ be a valuation path. If $u_i \neq u_{i+1}$ for all $i < n$, then $va(P)$ is a contractive valuation algebra.*

Proof. Let $i \leq n + 1$ and $a \in \mathcal{A}$, then we need to show that $a/(a \bullet i) = a/i$ and $a \bullet a \bullet i = a \bullet i$ in $va(P)$. If $a \bullet i = i$, then we are done. Thus assume that $a \bullet i = i + 1$, in which case $i \leq n$ and $u_i = a$. Since $u_{i+1} \neq u_i = a$ we get $a \bullet (i + 1) = i + 1$. Also, it is not hard to see that $i \leq \text{last}(a, i + 1) < i + 1$, but then $i = \text{last}(a, i + 1)$, and therefore $a/(i + 1) = b_i = a/i$. \square

Based on this, our first move will be to reduce or ‘contract’ a valuation path where some subsequent atoms are equal, to a corresponding valuation path where all subsequent atoms are different. This is not that difficult; whenever we find two subsequent segments with identical atoms, we omit one of them. More formally, we can define the contraction of a valuation path as follows:

Definition 4.24. Let P be a valuation path. We define the *contraction* of P , denoted $cn(P)$, by

$$\begin{aligned} cn(\epsilon) &= \epsilon & cn((u, b) \cdot Q) &= (u, b) \cdot cn_u(Q) \\ cn_a(\epsilon) &= \epsilon & cn_a((u, b) \cdot Q) &= \begin{cases} (u, b) \cdot cn_u(Q) & \text{if } u \neq a \\ cn_a(Q) & \text{otherwise} \end{cases} \end{aligned}$$

where cn_a is defined as above for each $a \in \mathcal{A}$.

Clearly, $cn(cn(P)) = cn(P)$ for all valuation paths P . Example 4.25 shows that in general, $cn(P \cdot Q) \neq cn(P) \cdot cn(Q)$. However, $cn(P \cdot Q) = cn(cn(P) \cdot cn(Q))$ for all P and Q . These observations are illustrated by Figure 4.6.

Example 4.25. For instance, let $P = \langle (a, T), (a, F), (b, T) \rangle$ and $Q = \langle (b, T) \rangle$, then $cn(P) = \langle (a, T), (b, T) \rangle$, $cn(Q) = Q$ and $cn(P \cdot Q) = cn(cn(P) \cdot cn(Q)) = cn(P)$.

It can be easily checked that the contraction norm defined by $\|P\| = |cn(P)|$ is a norm. We define the norm-based constructor cva as a special case of va .

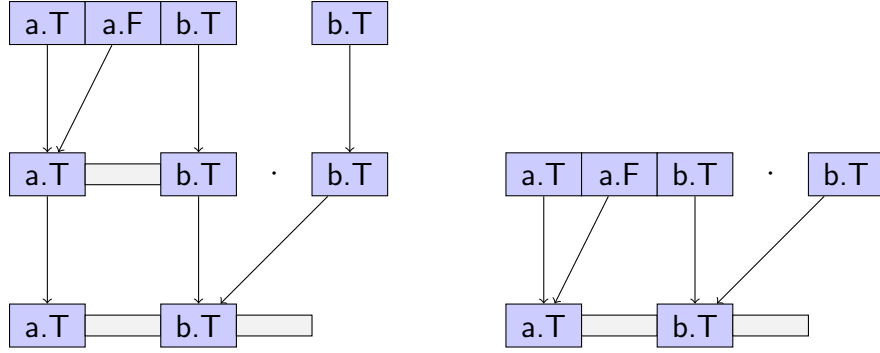


Figure 4.6.: A schematic depiction of Example 4.25.

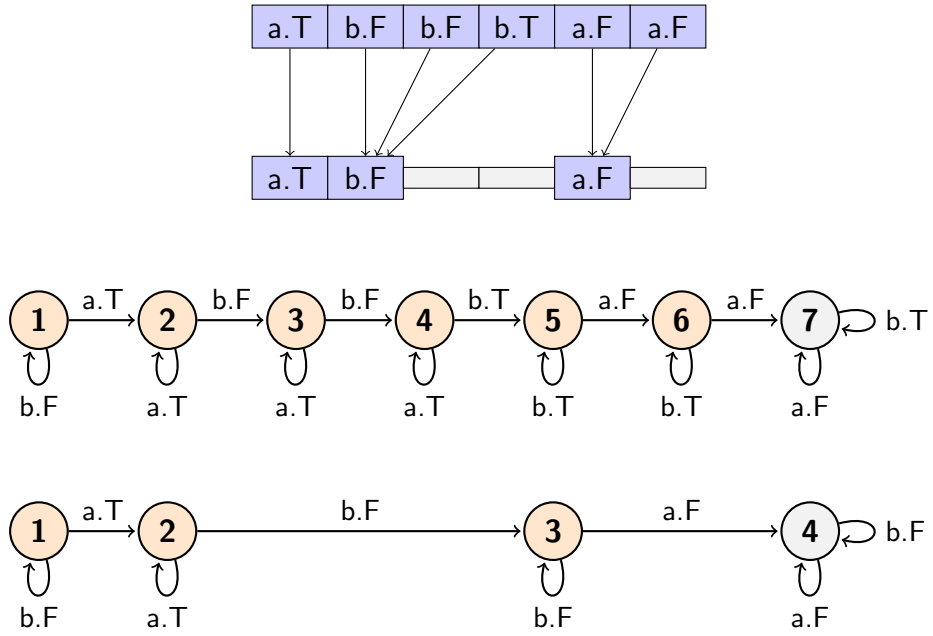


Figure 4.7.: The valuation algebras $va(P)$ and $cva(P)$ strongly resemble P and $cn(P)$ respectively; here, $P = \langle (a, T), (b, F), (b, F), (b, T), (a, F), (a, F) \rangle$.

Definition 4.26. Let P be a valuation path. We define $\text{cva}(P) = \text{va}(\text{cn}(P))$.

The relation between va and cva is illustrated by Figure 4.7. Of course, if $\text{cn}(P) = P$, then $\text{cva}(P) = \text{va}(P)$. Proposition 4.23 tells us that cva constructs valuation algebras that are all in **cr**.

Lastly, we need to construct valuation algebras that are in **st**. We could again change va to do this, but here we take a simpler approach. We construct trivial valuation algebras using a norm-based constructor for the trivial norm. By Proposition 3.10, the valuation algebras constructed this way are static.

Definition 4.27. Let $P = \langle (u_1, b_1), \dots, (u_n, b_n) \rangle$ be a valuation path. We define $\text{sva}(P)$ as the valuation algebra $(\{1\}, /, \bullet)$ where $a/1 = \text{T}$ for $a \in \mathcal{A}$ if and only if there exists $i \leq n$ such that $u_i = a$ and $b_i = \text{T}$, and $a \bullet 1 = 1$ for all $a \in \mathcal{A}$.

4.4. Satisfiability and Path-Satisfiability

In the previous section, we have created three norm-based constructors that create valuation algebras based on valuation paths. In this section, we need to prove that these valuation algebras do what they are intended to do. That is, we need to prove that if x is a formula and P a valuation path such that $P : \text{se}(x)$ is defined, then x must evaluate to the truth value $P : \text{se}(x)$ in the valuation algebra constructed by va and, under certain circumstances, also the valuation algebras constructed cva and sva .

The phrase “under certain circumstances” is definitely necessary. Suppose for instance that for every valuation path P such that $P : \text{se}(x) = \text{T}$, x evaluates to T in the valuation algebra $\text{sva}(P)$. This would mean that every x that is path-satisfiable, is satisfiable with respect to **st**. If we look at Figure 4.3, this results in all five satisfiabilities being the same; this is clearly not the case, as the formula $a \wedge \neg a$ is satisfiable with respect to **fr**, but not to **st**.

The exact ‘circumstances’ are these: x must evaluate to $P : \text{se}(x)$ in $\text{cva}(P)$ if P is repetition-proof, and in $\text{sva}(P)$ if P is memorizing. The following Lemma tells us exactly what we need to prove.

Lemma 4.28. *Let x be a formula and P a valuation path such that $P : \text{se}(x)$ is defined. Then:*

- a. $x/1 = P : \text{se}(x)$ in $\text{va}(P)$;
- b. if P is repetition-proof, then $x/1 = P : \text{se}(x)$ in $\text{cva}(P)$;
- c. if P is memorizing, then $x/1 = P : \text{se}(x)$ in $\text{sva}(P)$.

The proof of this lemma is based on an earlier remark: norm-based constructors such as va are somehow ‘invariant under concatenation’. This meant that if we take a small chunk of a valuation path, say P_2 as part of $P_1 \cdot P_2 \cdot P_3$, then the valuation algebra $\text{va}(P_2)$ is in a sense ‘embedded’ in $\text{va}(P_1 \cdot P_2 \cdot P_3)$. In particular, this is supposed to mean that

if $P_2 : \text{se}(x)$ is defined, then not only does x evaluate to $P_2 : \text{se}(x)$ in $\text{va}(P_2)$, but also somewhere in the larger valuation algebra $\text{va}(P_1 \cdot P_2 \cdot P_3)$.

We will formalise these notions by defining that a formula is “regular” if it has such behaviour for all valuation paths $P_1 \cdot P_2 \cdot P_3$. We then proceed to prove by induction that all formulas are regular.

Definition 4.29. Let \mathbf{u} be a norm-based constructor for a norm $\|\cdot\|$. Let \mathcal{C} be a collection of valuation paths. A formula x is *regular* on \mathbf{u} with respect to \mathcal{C} if for all paths $P = P_1 \cdot P_2 \cdot P_3$ in \mathcal{C} such that $P_2 : \text{se}(x)$ is defined, the following holds:

$$x / (\|P_1\| + 1) = P_2 : \text{se}(x) \quad \text{and} \quad x \bullet (\|P_1\| + 1) = \|P_1 \cdot P_2\| + 1$$

in the valuation algebra $\mathbf{u}(P)$.

As a special case, we can take $P_1 = \epsilon = P_3$ to obtain that $x/1 = P : \text{se}(x)$ in $\mathbf{u}(P)$ for all P in \mathcal{C} , for all x that are regular on \mathbf{u} with respect to \mathcal{C} . Thus, we are now left to prove that all formulas are regular on va with respect to the collection of all valuation paths, regular on cva w.r.t. the collection of contractive paths, and regular on sva w.r.t. the collection of memorizing paths. We prove this by induction. To avoid unnecessary repetition, we use the following proposition.

Proposition 4.30. *Let \mathbf{u} be a norm-based constructor and let \mathcal{C} be a collection of valuation paths. If every formula of the form a where $a \in \mathcal{A}$ is regular on \mathbf{u} with respect to \mathcal{C} , then so are all other formulas.*

Proof. We will prove that all formulas are regular on \mathbf{u} with respect to \mathcal{C} by induction on the complexity of the formula. Since the atoms are part of the premise, we need to consider the formulas of the form \top , $\neg x$ and $x \triangleleft y$.

First, note that $\text{se}(\top) = \top$, thus if $P = P_1 \cdot P_2 \cdot P_3$ is a path in \mathcal{C} with $P_2 : \text{se}(\top)$ is defined, then $P_2 = \epsilon$ and $P_2 : \text{se}(\top) = \top$. As with any other valuation algebra, $\top / (\|P_1\| + 1) = \top$ and $\top \bullet (\|P_1\| + 1) = \|P_1\| + 1 = \|P_1 \cdot \epsilon\| + 1$ in $\mathbf{u}(P)$. Therefore, \top is regular.

Let x be regular. Let $P = P_1 \cdot P_2 \cdot P_3$ in \mathcal{C} with $P_2 : \text{se}(\neg x)$ is defined. By Proposition 4.5 we get that $P_2 : \text{se}(\neg \neg x) = \neg(P_2 : \text{se}(\neg x))$, and because $\text{se}(\neg \neg x) = \text{se}(x)$ we have $P_2 : \text{se}(x) = \neg(P_2 : \text{se}(\neg x))$. Since x is regular, we find

$$\begin{aligned} (\neg x) / (\|P_1\| + 1) &= \neg(x / (\|P_1\| + 1)) = \neg(P_2 : \text{se}(x)) = P_2 : \text{se}(\neg x), \\ (\neg x) \bullet (\|P_1\| + 1) &= x \bullet (\|P_1\| + 1) = \|P_1 \cdot P_2\| + 1, \end{aligned}$$

in $\mathbf{u}(P)$. Thus $\neg x$ is regular.

Finally, let x and y be regular. Let $P = P_1 \cdot P_2 \cdot P_3$ in \mathcal{C} with $P_2 : \text{se}(x \triangleleft y)$ is defined. If we take $X = \text{se}(x)$, $Y = \text{se}(y)$ and $Y' = \text{F}$, then we get $\text{se}(x \triangleleft y) = X[\top \mapsto Y, \text{F} \mapsto Y']$. Now we can apply Proposition 4.6 to obtain paths R and Q such that $P_2 = R \cdot Q$, $R : \text{se}(x)$ is defined and

$$P_2 : \text{se}(x \triangleleft y) = \begin{cases} Q : \text{se}(y) & \text{if } R : \text{se}(x) = \top \\ Q : \text{F} & \text{otherwise} \end{cases}$$

Note that we can write P as $P_1 \cdot R \cdot (Q \cdot P_3)$. Since $R : \text{se}(x)$ is defined and x is regular, we get that $x/(||P_1|| + 1) = R : \text{se}(x)$ and $x \bullet (||P_1|| + 1) = ||P_1 \cdot R|| + 1$. This gives us

$$\begin{aligned} (x \wedge y)/(||P_1|| + 1) &= \begin{cases} y/(x \bullet (||P_1|| + 1)) & \text{if } x/(||P_1|| + 1) = \text{T} \\ \text{F} & \text{otherwise} \end{cases} \\ &= \begin{cases} y/(||P_1 \cdot R|| + 1) & \text{if } R : \text{se}(x) = \text{T} \\ \text{F} & \text{otherwise} \end{cases} \end{aligned}$$

and

$$\begin{aligned} (x \wedge y) \bullet (||P_1|| + 1) &= \begin{cases} y \bullet (x \bullet (||P_1|| + 1)) & \text{if } x/(||P_1|| + 1) = \text{T} \\ x \bullet (||P_1|| + 1) & \text{otherwise} \end{cases} \\ &= \begin{cases} y \bullet (||P_1 \cdot R|| + 1) & \text{if } R : \text{se}(x) = \text{T} \\ ||P_1 \cdot R|| + 1 & \text{otherwise} \end{cases} \end{aligned}$$

We will show that $(x \wedge y)/(||P_1|| + 1) = P_2 : \text{se}(x \wedge y)$ and $(x \wedge y) \bullet (||P_1|| + 1) = ||P_1 \cdot P_2|| + 1$ by considering both cases separately.

Suppose that $R : \text{se}(x) = \text{T}$. We can rewrite P once more, this time as $(P_1 \cdot R) \cdot Q \cdot P_3$. Because y is regular as well and $Q : \text{se}(y) = P_2 : \text{se}(x \wedge y)$ is defined, we now get $y/(||P_1 \cdot R|| + 1) = Q : \text{se}(y)$ and $y \bullet (||P_1 \cdot R|| + 1) = ||P_1 \cdot R \cdot Q|| + 1 = ||P_1 \cdot P_2|| + 1$. Suppose otherwise, then we must have $Q = \epsilon$ since $Q : \text{F} = P_2 : \text{se}(x \wedge y)$ is defined. Thus we get $Q : \text{F} = \text{F}$, and $||P_1 \cdot R|| + 1 = ||P_1 \cdot R \cdot Q|| + 1$. In either case, we have shown that $x \wedge y$ is regular.

This concludes the inductive proof. \square

We have now done a lot of hard work already. We still need to prove that all formulas $a \in \mathcal{A}$ are regular with respect to the appropriate constructors and collections. We will prove each of the three parts separately, starting with va .

Proof (Lemma 4.28a). Let $a \in \mathcal{A}$, then $\text{se}(a) = \text{T} \trianglelefteq a \trianglerighteq \text{F}$. If $P = P_1 \cdot P_2 \cdot P_3$ is a path with $P_2 : \text{se}(a)$ is defined, then $P_2 = \langle (a, b) \rangle$ for some $b \in \{\text{T}, \text{F}\}$, and $P_2 : \text{se}(a) = b$. If we write $P = \langle p_1, \dots, p_n \rangle$ and $k = |P_1| + 1$, then $p_k = (a, b)$ and $\text{last}(a, k) = k$. From this, we get $a/(|P_1| + 1) = b_k = b$ and $a \bullet (|P_1| + 1) = k + 1 = |P_1| + |P_2| + 1 = |P_1 + P_2| + 1$.

Now we have shown that every $a \in \mathcal{A}$ is regular on va with respect to the collection of all valuation paths. By Proposition 4.30, every formula is regular. So, if we take a formula x and a valuation path P such that $P : \text{se}(x)$ is defined, then we choose $P_1 = \epsilon$, $P_2 = P$ and $P_3 = \epsilon$ to obtain, with $||\epsilon|| + 1 = 1$, that $x/1 = P : \text{se}(x)$. \square

As cva is based on va , the proof of the second part of Lemma 4.28 is based on the previous proof. However, it is a bit more complex. The difficulty lies where we want to evaluate an atom $a \in \mathcal{A}$ that occurs in a path $P_1 \cdot \langle (a, b) \rangle \cdot P_3$ for some valuation paths P_1 and P_3 and some $b \in \{\text{T}, \text{F}\}$. Normally, we know which valuation the atom is evaluated in, but if P_1 ended with a , this valuation will be “contracted away” in a sense. For this reason, we will need a case distinction that depends on the last segment of P_1 , and we will need to use the fact that cva is based on contraction.

Proof (Lemma 4.28b). Let $a \in \mathcal{A}$. If $P = P_1 \cdot P_2 \cdot P_3$ is a repetition-proof path with $P_2 : \text{se}(a)$ is defined, then $P_2 = \langle (a, b) \rangle$ for some $b \in \{T, F\}$, and $P_2 : \text{se}(a) = b$. We write $\text{cn}(P) = \langle p'_1, \dots, p'_{n'} \rangle$, $m = |\text{cn}(P_1)|$ and $k = |\text{cn}(P_1 \cdot P_2)|$, and we note that $m \leq k \leq m + 1$ and $k \geq 1$. Also, $\text{cn}(P_1) = \langle p'_1, \dots, p'_m \rangle$, $\text{cn}(P_1 \cdot P_2) = \langle p'_1, \dots, p'_k \rangle$ and $p'_k = (a, b')$ for some $b' \in \{T, F\}$. Note that here, last is given by the definition of $\text{cva}(P) = \text{va}(\text{cn}(P))$.

Suppose $m = 0$, then $\text{cn}(P_1) = \epsilon$ and $P_1 = \epsilon$, which means $\text{cn}(P_1 \cdot P_2) = P_2$ and $k = 1$. Now we easily see $\text{last}(a, 1) = 1$ so $a/1 = b$, and $a \bullet 1 = 1 + 1 = |\text{cn}(P_1 \cdot P_2)| + 1$.

Suppose $m > 0$ and $u'_m \neq a$, then $\text{cn}(P_1 \cdot P_2) = \text{cn}(P_1) \cdot P_2$, which means $k = m + 1$ and $p'_k = (a, b)$ so $b' = b$. We find $\text{last}(a, m + 1) = k$ thus $a/k = b'_k = b$, and $u'_k = a$ thus $a \bullet k = k + 1 = |\text{cn}(P_1 \cdot P_2)| + 1$.

Suppose $m > 0$ and $u'_m = a$, then $\text{cn}(P_1 \cdot P_2) = \text{cn}(P_1)$ and $k = m$ and $p'_m = (a, b')$. By construction $u'_{m+1} \neq u'_m = a$, so $\text{last}(a, m + 1) = m$, thus $a/(m + 1) = b'_m$ and $a \bullet (m + 1) = m + 1 = |\text{cn}(P_1 \cdot P_2)| + 1$. By definition of $\text{cn}(P)$, we know that $p'_m = p_i$ for some $i \leq n$; that is, if we write $P_1 = \langle p_1, \dots, p_{n_1} \rangle$, $P_2 = \langle p_{n_1+1} \rangle$ and $P_3 = \langle p_{n_1+2}, \dots, p_n \rangle$, then there is some $i \leq n_1$ such that $p_i = p'_m = (a, b')$ and $p_j = (a, b_j)$ for $i \leq j \leq n_1 + 1$. Because P is repetition-proof, $b' = b_j$ for all $1 \leq j \leq n_1 + 1$, so $b' = b_{n_1+1} = b$. Now we find $a/(m + 1) = b$.

Now we have shown that every $a \in \mathcal{A}$ is regular on cva with respect to the collection of repetition-proof valuation paths. By Proposition 4.30, we are done. \square

Finally, the third part of Lemma 4.28. This part is arguably the easiest, as the valuation algebras constructed by sva are trivial.

Proof (Lemma 4.28c). Let $a \in \mathcal{A}$. If $P = P_1 \cdot P_2 \cdot P_3$ is memorizing with $P_2 : \text{se}(a)$ is defined, then $P_2 = \langle (a, b) \rangle$ where $b = P_2 : \text{se}(a)$. If we write $P = \langle p_1, \dots, p_n \rangle$ and $k = |P_1| + 1$, then $P_2 = \langle p_k \rangle$. If $b = T$, then $a/1 = T$. If not then $p_k = (a, F)$ and there can be no other i with $p_i = (a, T)$, since P is memorizing. This means $a/1 = F$. Also, $a \bullet 1 = 1$.

Now we have shown that every $a \in \mathcal{A}$ is regular on sva with respect to the collection of memorizing valuation paths. By Proposition 4.30, we are done. \square

With Lemma 4.28 proven, we can state the following result:

Theorem 4.31. *Let x be a formula.*

- a. *If $\text{PathSat}_{fr}(x)$, then $\text{SAT}_{fr}(x)$.*
- b. *If $\text{PathSat}_{rp}(x)$, then $\text{SAT}_{cr}(x)$.*
- c. *If $\text{PathSat}_{mem}(x)$, then $\text{SAT}_{st}(x)$.*

Proof. Let P be a valuation path such that $P : \text{se}(x) = T$. By Lemma 4.28a, $x/1 = T$ in $\text{va}(P)$. As any valuation algebra is in **fr**, this means $\text{SAT}_{fr}(x)$.

If P is also repetition-proof (resp. memorizing), then Lemma 4.28b (resp. 4.28c) tells us that $x/1 = T$ in $\text{cva}(P)$ (resp. $\text{sva}(P)$). By Proposition 4.23 (resp. Proposition 3.10), this valuation algebra is in **cr** (resp. **st**), and this means $\text{SAT}_{cr}(x)$ (resp. $\text{SAT}_{st}(x)$). \square

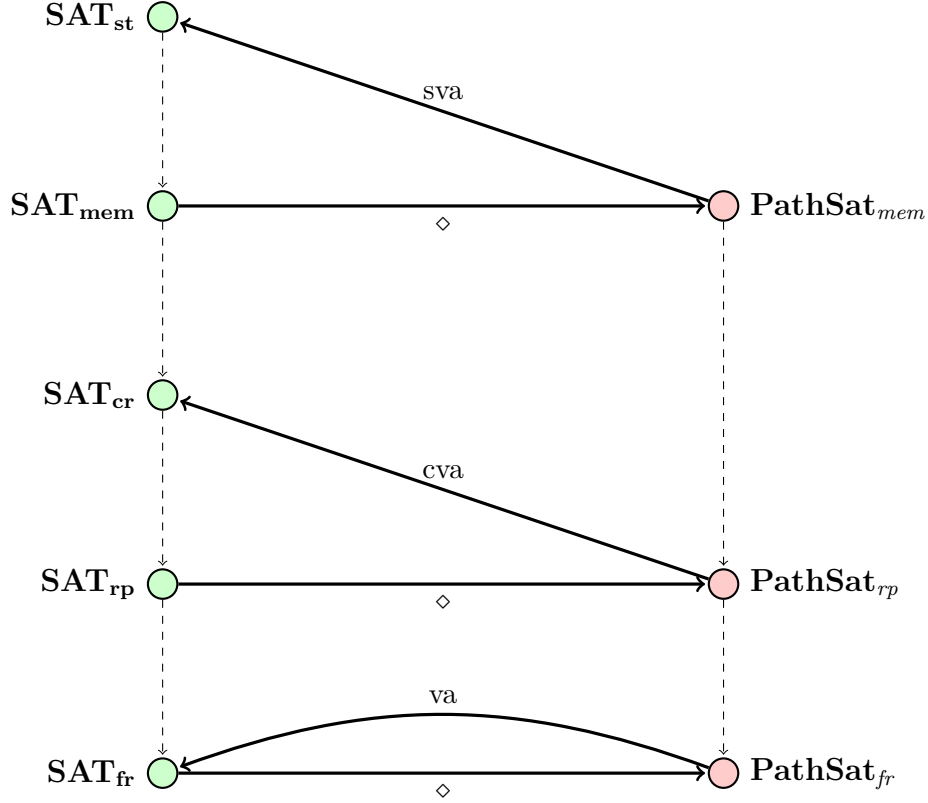


Figure 4.8.: An updated overview, based on Figure 4.3. The three thick arrows labeled va , cva and sva are given by Theorem 4.31.

This theorem allows us to complete our overview on the connections between satisfiability and path-satisfiability, as illustrated by Figure 4.8. From this figure, another important result can be deduced: our five short-circuit logics only define three different types of satisfiability. The causes and implications of this are discussed in Chapter 6, but it is already shown by the following corollary.

Corollary 4.32. *For all x , $\mathbf{SAT}_{rp}(x) \iff \mathbf{SAT}_{cr}(x)$ and $\mathbf{SAT}_{mem}(x) \iff \mathbf{SAT}_{st}(x)$.*

Proof. By Proposition 3.13, we already had one part (i.e. \Leftarrow) of both equivalences. Combining Theorem 4.31 with Theorem 4.18 now gives us $\mathbf{SAT}_{rp}(x) \Rightarrow \mathbf{SAT}_{cr}(x)$ and $\mathbf{SAT}_{mem}(x) \Rightarrow \mathbf{SAT}_{st}(x)$. \square

5. Implementation in Haskell

The implementation is done in Haskell, although the methods described can be adapted to most programming languages. The choice for Haskell is based mostly on ease of development, as its syntax is very reminiscent of mathematical language and therefore very suitable for satisfiability testing and theorem-proving. Inspiration on implementing formulas in Haskell was taken from [10].

5.1. Formulas, Trees and Paths

5.1.1. Formula

The data type `Formula` implements formulas.

```
data Formula    = Lit Atom
                | Const Bool
                | Neg Formula
                | Con Formula Formula
                | Dis Formula Formula
    deriving (Eq)
```

The type `Atom` is synonymous with `String`. Note that this implementation considers `F` and $x \vee y$ to be formulas, and not abbreviations as in Chapter 2. This is because the formula $\neg(\neg x \wedge \neg y)$ requires more memory to store and takes more time to process. These optimisations are valued higher than the lack of redundant code. When printed, the textual symbols `T`, `F`, `!`, `&&` and `||` are used to represent `T`, `F`, \neg , \wedge and \vee respectively.

5.1.2. Tree

The data type `Tree` implements trees. When printed, the textual symbols `T`, `F`, `<` and `>` are used to represent `T`, `F`, \trianglelefteq and \trianglerighteq respectively.

```
data Tree      = Leaf Bool
                | Branch Tree Atom Tree
    deriving (Eq)
```

Trees grow exponentially as the number of atoms grow. For each ‘junction’, i.e. either \wedge or \vee , approximately half (either all `T` or all `F`) of the leaves in the first tree are replaced by new trees. As can be shown by a simple inductive proof, the number of junctions in a constant-free formula is one less than the number of atoms in that formula. Thus, the

memory size of a tree is doubled for each atom added. This means the memory required to store the se-tree of a constant-free formula with n atoms is estimated to be $\mathcal{O}(2^n)$. Formulas with constants can have smaller se-trees, e.g. $\text{se}(F \wedge (a \vee b)) = F$. The number of leaves in the se-tree of a formula with n atoms is $\mathcal{O}(2^n)$ as well.

The use of data pointers to reduce the memory requirements was considered, such as storing each branch only once and replacing leaves with pointers to branches instead of copies of trees. However, this might be more suitable for an implementation in a low level language such as C, as opposed to Haskell. Alternatively, it is possible to enumerate all possible formulas over a countably infinite set of atoms \mathcal{A} , and to use numbers to represent formulas instead of data structures. Again, however, this would be more suitable for an implementation focused on execution speed instead of experimentation and readability.

5.1.3. Path

Valuation paths are implemented by the type `Path`. Paths are simply printed as is.

```
type Path = [(Atom, Bool)]
```

The function `isPathRepProof` checks if the given path is repetition-proof by recursively comparing each element of the path with the element directly after it; its complexity is $\mathcal{O}(n)$, where n is the length of the path.

```
isPathRepProof :: Path -> Bool
isPathRepProof [] = True
isPathRepProof (hd : rest) = check hd rest
  where
    check (a, b) p = case p of
      []      -> True
      (h : t) -> if (fst h) == a
        then if (snd h) == b
          then check h t
          else False
        else check h t
```

The function `isPathMemorizing` checks if the given path is memorizing by keeping a list of all the bindings made; this has a worst case complexity of $\mathcal{O}(n^2)$.

```
isPathMemorizing path = check [] path
  where
    check rs p = case p of
      []      -> True
      (h : t) -> case lookup (fst h) rs of
        (Just b)    -> if (snd h) == b
          then check rs t
          else False
        (Nothing)   -> check (h : rs) t
```


The function `checkPath` chooses the appropriate function for the given logic.

```
checkPath :: Logic -> Path -> Bool
checkPath FSCL      = (\ _ -> True)
checkPath RPSCL     = isPathRepProof
checkPath CSCL      = isPathRepProof
checkPath MSCL      = isPathMemorizing
checkPath SSCL      = isPathMemorizing
```

5.1.4. Logic

The data type `Logic` simply consists of five constants; one for each of the five short-circuit logics described in Section 2.3.

5.1.5. Result

The data type `Result` is used by various functions as a generic piece of error-handling specific to this implementation. In particular, such functions can return **Yes**, **No** and **Unknown**. The latter is used when satisfiability testers for one logic are used on a formula in another logic, which may return render the testing inconclusive.

5.1.6. Normal Form

As mentioned in Section 2.5, a function f exists which maps formulas to normal form equivalents. In Section 2.5, it is discussed that normal forms resemble se-trees. In Section 4.2, this is expanded upon by two corollaries, Corollary 4.11 and Corollary 4.12, that suggest that the function f can be used to determine path-satisfiability. Unfortunately, just like se-trees (Section 5.1.2), the normal forms grow exponentially in size; for a formula containing n atoms, which is thus of size $\mathcal{O}(n)$, the normal form formula created by applying f is of size $\mathcal{O}(2^n)$.

5.2. Satisfiability Testers

The data type `SatTester` implements satisfiability testers for short-circuit logic.

```
type SatTester = Logic -> Formula -> Result
```

A `SatTester` is a function that determines if a formula x is path-satisfiable with regards to a certain logic. A `SatTester` should result either **Yes p** if p is a path that satisfies the formula, **No** if the formula is not path-satisfiable, or **Unknown** if it is not able to conclude either answer with certainty. Five `SatTesters` are implemented: `SatBruteControl`, `SatBruteForce`, `SatDirect`, `SatOpen`, and `SatBoolean`.

5.2.1. SatBruteControl

SatBruteControl is a $\mathcal{O}(2^n)$ satisfiability tester for all logics. It tries to construct a path based on the se-tree of the formula, by searching the leaves for T. If it is found, the path created along the way is checked using the `checkPath` function.

```
findValidSolution :: Logic -> Tree -> Path -> (Bool, Path)
findValidSolution lg et p = case et of
  (Leaf True)    -> (checkPath lg p, p)
  (Leaf False)   -> (False, p)
  (Branch l c r) ->
    let
      soll = findValidSolution lg l (p ++ [(c, True)])
      solr = findValidSolution lg r (p ++ [(c, False)])
    in if fst soll
      then (fst soll, (c, True) : (snd soll))
      else (fst solr, (c, False) : (snd solr))
```

SatBruteControl continues searching until an appropriate T leaf is found, thus in the worst case this involves building and searching the entire tree, which is a $\mathcal{O}(2^n)$ operation. Each time a leaf is found, the constructed path has to be checked, which is a mere $\mathcal{O}(n^2)$ operation in the worst case (see Section 5.1.3).

5.2.2. SatBruteForce

SatBruteForce is also a $\mathcal{O}(2^n)$ satisfiability tester for all logics. It is a minor improvement over SatBruteControl. For MSCL and SSCL they coincide. For FSCL, the path is not checked since any path is valid, which eliminates the need to carry the path down the recursion.

```
findAnySolution :: Tree -> (Bool, Path)
findAnySolution et = case et of
  (Leaf True)    -> (True, [])
  (Leaf False)   -> (False, [])
  (Branch l c r) ->
    let
      soll = findAnySolution l
      solr = findAnySolution r
    in if fst soll
      then (fst soll, (c, True) : (snd soll))
      else (fst solr, (c, False) : (snd solr))
```

For RPSCL and CSCL, the path that is created while searching for T leaves is directly checked to be repetition-proof by carrying the last encountered atom down the recursion.

```

findRepProofSolution :: Tree -> Maybe (Atom, Bool) -> (Bool, Path)
findRepProofSolution et m = case et of
  (Leaf True)      -> (True, [])
  (Leaf False)     -> (False, [])
  (Branch l c r)   -> case m of
    (Nothing)      ->
      let
        soll = findRepProofSolution l (Just (c, True))
        solr = findRepProofSolution r (Just (c, False))
      in if fst soll
        then (fst soll, (c, True) : (snd soll))
        else (fst solr, (c, False) : (snd solr))
    (Just (a, b))   -> if (a == c)
      then
        let
          p = (if b then l else r)
          solp = findRepProofSolution p (Just (c, b))
        in (fst solp, (c, b) : (snd solp))
      else findRepProofSolution et (Nothing)

```

Both methods still require most of the se-tree to be searched, so `SatBruteForce` is $\mathcal{O}(2^n)$ in the worst case for all logics.

5.2.3. SatDirect

`SatDirect` is a $\mathcal{O}(n)$ satisfiability tester for FSCL, but can be used for other logics as well. If no path is found, then this means the formula is not satisfiable for any logic. If a path is found, then either the path is usable within the logic and `Yes` is returned, or it is not, in which case `Unknown` is returned.

It is based on remarks made in [2], which state

$$\begin{array}{ll}
\mathbf{SAT}_{\mathbf{fr}}(T) & \neg \mathbf{FAL}_{\mathbf{fr}}(T) \\
\mathbf{SAT}_{\mathbf{fr}}(a) & \mathbf{FAL}_{\mathbf{fr}}(a) \\
\mathbf{SAT}_{\mathbf{fr}}(\neg x) \Leftrightarrow \mathbf{FAL}_{\mathbf{fr}}(x) & \mathbf{FAL}_{\mathbf{fr}}(\neg x) \Leftrightarrow \mathbf{SAT}_{\mathbf{fr}}(x) \\
\mathbf{SAT}_{\mathbf{fr}}(x \circlearrowleft y) \Leftrightarrow \mathbf{SAT}_{\mathbf{fr}}(x) \wedge \mathbf{SAT}_{\mathbf{fr}}(y) & \mathbf{FAL}_{\mathbf{fr}}(x \circlearrowleft y) \Leftrightarrow \mathbf{FAL}_{\mathbf{fr}}(x) \vee \mathbf{FAL}_{\mathbf{fr}}(y)
\end{array}$$

where a ranges over \mathcal{A} . This allows us to tell if a formula is satisfiable or not.

The function `sat` determines if a formula is satisfiable, and supplies a satisfying path if it is. If it is not, the supplied path is discarded.

```

sat :: Formula -> (Bool, Path)
sat (Const True)      = (True, [])
sat (Const False)     = (False, [])
sat (Lit a)            = (True, [(a, True)])

```

```

sat (Neg f)                = fal f
sat (Con f1 f2)            = if fst (sat f1)
  then (fst (sat f2), snd (sat f1) ++ snd (sat f2))
  else sat f1
sat (Dis f1 f2)            = if fst (sat f1)
  then sat f1
  else (fst (sat f2), snd (sat f1) ++ snd (sat f2))

```

The definition of `fal` is similar, but the roles of T and F, and of \wedge and \vee are swapped; it determines if a formula is falsifiable instead.

Each atom is only visited once. As FSCL imposes no restrictions, the decision at each atom is trivial: ‘true’ if we want the formula to be satisfiable, or ‘false’ if we want it to be falsifiable. Note that for a formula $x \wedge y$, or `Con x y` in the code, we first try satisfiability for x . If x is satisfiable, we concatenate this result with the result of y . If it is not, then $x \wedge y$ cannot be satisfiable, so we are done. In this manner, `SatDirect` itself uses short-circuit evaluation to determine satisfiability.

5.2.4. SatOpen

`SatOpen` is a $\mathcal{O}(n)$ satisfiability tester for RPSCL and CSCL, but can be used for other logics as well. If no path is found, then `No` is returned for logics other than FSCL. If a path is found, then either `Yes` or `Unknown` is returned based on the usability within the given logic.

It is based on `SatDirect`. Again, each atom only needs to be visited once, but this time the decision to make an atom either ‘true’ or ‘false’ is more complicated. As an example, a naive way to try to solve $\mathbf{SAT}_{\text{rp}}((a \vee b) \wedge \neg a)$ would do the following: first, we examine $a \vee b$. This can be made true by taking a to be true. Now, we proceed to $\neg a$, and discover that a must be false immediately afterwards. This is not allowed under the rules and regulations of rp-path-satisfiability, so we might wrongly assume that $(a \vee b) \wedge \neg a$ is not rp-path-satisfiable. Of course we can take the path $\langle (a, F), (b, T), (a, F) \rangle$ and show that it is even mem-path-satisfiable. The decision made when we first came across a must have been the wrong one, but this seems impossible to tell without knowing what lies ahead. This lack of knowledge of future events is one of the main obstacles in solving propositional satisfiability, where guessing (and backtracking in case the guess was wrong) is usually the only option.

In the case of short-circuit logic, the solution is much simpler: we simply work from the back to the front. The formula $\neg a$ is satisfiable, but only by taking a to be false. With this knowledge we try to make $a \vee b$ satisfiable. If we encounter a , then we know we must take a to be false, preventing the creation of paths that are not repetition-proof. In the code, this knowledge is represented by a ‘guard’:

```

type Guard = (Maybe Atom, Maybe Path, Maybe Path)

```

A guard is either empty (if no atoms have been assigned a value) or it contains an atom. If it does, then it must contain either a path where that atom is true, one where that atom is false, or both. These guards are used when determining if an atom is satisfiable.

```

(Lit a) -> case g of
  (Just _, Just p, _)
    -> (Just (Just a, Just ((a, True) : p), Nothing))
  (Just u, Nothing, Just p)
    -> if u == a
      then (Nothing)
      else (Just (Just a, Just ((a, True) : p), Nothing))
  (Nothing, Nothing, Nothing)
    -> (Just (Just a, Just [(a, True)], Nothing))
  -
    -> error ("illegal u-guard" ++ show g)

```

If the guard is empty, contains a different atom, or contains a path starting with (a, T) , the atom a can be made true without a problem. If the guard ‘forces’ a to be false, then it cannot also be true, so the formula is not satisfiable. Once a new atom is assigned a value, a new guard is made, and the old guard can be discarded. This allows the algorithm to remain $\mathcal{O}(n)$.

In the introduction to this thesis, we explained that a formula $x \wedge y$ can only be true if both x and y are true, and that knowing x is false is enough to determine that $x \wedge y$ is also false. But of course the same holds if we know that y is false. If we want to know if $x \wedge y$ is rp-path-satisfiable, we first determine if y is. If it is not, then we do not need to consider the satisfiability of x . In this way, SatOpen uses short-circuit evaluation as well, but right-sequentially instead of left-sequentially.

5.2.5. SatBoolean

SatBoolean is a satisfiability tester for MSCL and SSCL.

In Section 3.2, it is discussed that satisfiability for SSCL coincides with propositional satisfiability. See Figure 5.1 for an overview of this. Propositional satisfiability, also called boolean satisfiability, has already been the subject of many research papers, and has been proven to be NP-complete by Stephen Cook in 1971, as discussed in [6] and [7]; therefore, a new implementation would be mostly pointless. SatBoolean instead provides a wrapper function for a boolean satisfiability solver, an implementation of propositional satisfiability that uses the Davis-Putnam-Logemann-Loveland algorithm defined in [8] and [9]. Its complexity is the same as that of the DPLL algorithm, thus $\mathcal{O}(2^n)$ worst case.

First, the formula is translated to the correct format. Then the imported SatSolver module is used to determine propositional satisfiability. If no solution is found, **No** is returned for the logics MSCL and SSCL; for other logics, **Unknown** is returned. If a solution is found, the function `makePath` uses it to construct a valuation path.

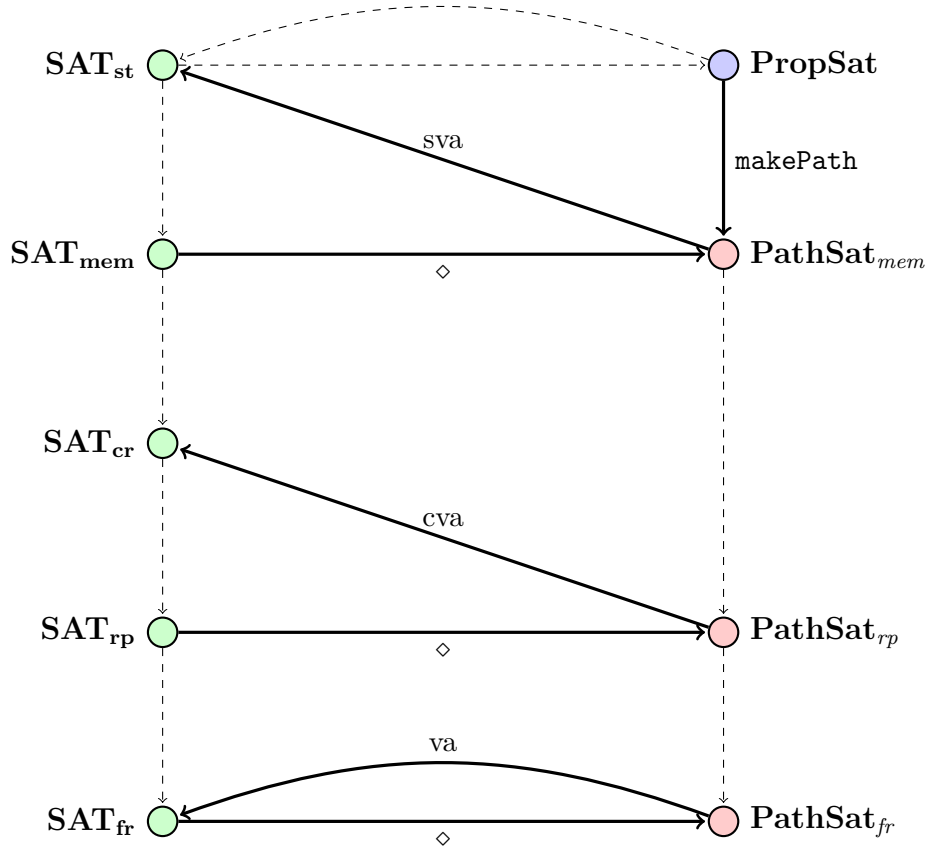


Figure 5.1.: A final overview, based on Figure 4.8. The blue node represents propositional satisfiability, as mentioned in Section 2.3 and Section 3.2. The two dashed arrows connected to it are given by Corollary 3.16. The thick arrow labeled **makePath** leading down from it is discussed in Section 5.2.5.

6. Conclusion

This thesis set out to define evaluation and satisfiability for each of the five short-circuit logics defined in [1]; FSCL (‘free short-circuit logic’), RPSCL (‘repetition-proof’), CSCL (‘contractive’), MSCL (‘memorizing’) and SSCL (‘static’). The desire to implement a program that could test this satisfiability lead to a different definition based on paths, which we called ‘path-satisfiability’. Three types of valuation paths were defined, corresponding to the terms ‘free’, ‘repetition-proof’ and ‘memorizing’.

A considerable portion of this thesis was spent proving the correspondences between the theoretically defined satisfiability and the path-satisfiability that was implemented. The result of this work was that what had appeared to be five types of satisfiability, one for each short-circuit logic, turned out to be only three types. It is proven that RPSCL and CSCL generate the same form of satisfiability, which is more restricted than that of FSCL, but less so than that of MSCL. The semantics of ‘repetition-proof’ restrict what truth values certain atoms can take, whereas the semantics of ‘contractive’ restrict what side-effects they can have. In the context of satisfiability, these side-effects only serve to alter truth values, so at that point the second restriction placed by CSCL is moot. A similar situation occurs between MSCL and SSCL; everything that can be achieved by using side-effects that obey the laws of MSCL, can also be achieved without the use of any side-effects. They too are proven to share their satisfiability.

In [1] and [2], it was already mentioned that static short-circuit logic was a sequential version of propositional logic. This was further discussed in this thesis; satisfiability for MSCL and SSCL both turned out to be equivalent to propositional (or ‘boolean’) satisfiability. Furthermore, it was shown that satisfiability for FSCL could be solved by using the short-circuit behaviour of the connectives \wedge and \vee ; satisfiability for RPSCL and CSCL could as well, but here this behaviour was right-sequential, as we worked from right to left instead.

Our implementation was more experimental in nature, and has room for many kinds of improvements. A reimplementing of the algorithms described in Chapter 5 could lead to more memory and time efficiency, thus to a more practical program. For analysing especially large formulas, such a new implementation could use parallelisation and even memoization to avoid doing double work.

Additionally, in the introduction it was mentioned that satisfiability for short-circuit logic could be applied to dead code detection. This might be something worth investigating in future papers.

Bibliography

- [1] Jan A. Bergstra, Alban Ponse, Daan J. C. Staudt, 2010, “Short Circuit Logic”, CoRR abs/1010.3674 v4, available at [arXiv:1010.3674v4](#) [cs.LO].
- [2] Jan A. Bergstra, Alban Ponse, 2011, “Proposition Algebra”, *ACM Transactions on Computational Logic, Volume 12 Issue 3*, Article No. 21.
- [3] Jan A. Bergstra, Alban Ponse, 2012, “Proposition Algebra and Short Circuit Logic”, *Fundamentals of Software Engineering*, 15-31.
- [4] Jan A. Bergstra, Alban Ponse, 2010, “On Hoare-McCarthy Algebras”, CoRR abs/1012.5059 v1, available at [arXiv:1012.5059v1](#) [cs.LO].
- [5] Jan A. Bergstra, Inge Bethke, Piet Rodenburg, 1995, “A propositional logic with 4 values: true, false, divergent and meaningless”, *Journal of Applied Non-Classical Logics, Volume 5 Issue 2*, 199-218.
- [6] Stephen A. Cook, 1971, “The complexity of theorem-proving procedures”, STOC '71 *Proceedings of the third annual ACM symposium on Theory of computing*, 151-158.
- [7] Thomas J. Schaefer, 1978, “The complexity of satisfiability problems”, STOC '78 *Proceedings of the tenth annual ACM symposium on Theory of computing*, 216-226.
- [8] Martin Davis, Hilary Putnam, 1960, “A Computing Procedure for Quantification Theory”, *Journal of the ACM, Volume 7 Issue 3*, 201-215.
- [9] Martin Davis, George Logemann, Donald Loveland, 1962, “A machine program for theorem-proving”, *Communications of the ACM, Volume 5 Issue 7*, 394-397.
- [10] Jan van Eijck, 2013, *Tutorial on Theorem Proving*, lecture material for the course “Functional Specification of Algorithm”.

A. Axioms of Short Circuit Logics

Remark. Note that because F and \vee are defined as abbreviations in this thesis, axioms A.1 and A.2 are technically not axioms but defining equations.

The system EqFSCL consists of the following 10 axioms:

$$F = \neg T \quad (A.1)$$

$$x \vee y = \neg(\neg x \wedge \neg y) \quad (A.2)$$

$$\neg\neg x = x \quad (A.3)$$

$$T \wedge x = x \quad (A.4)$$

$$x \wedge T = x \quad (A.5)$$

$$F \wedge x = F \quad (A.6)$$

$$(x \wedge y) \wedge z = x \wedge (y \wedge z) \quad (A.7)$$

$$x \wedge F = \neg x \wedge F \quad (A.8)$$

$$(x \wedge F) \vee y = (x \vee T) \wedge y \quad (A.9)$$

$$(x \wedge y) \vee (z \wedge F) = (x \vee (z \wedge F)) \wedge (y \vee (z \wedge F)) \quad (A.10)$$

The system EqRPSCL extends EqFSCL with the following axiom schemes, where a ranges over \mathcal{A} :

$$a \wedge (a \vee x) = a \wedge a \quad (A.11)$$

$$a \vee (a \wedge x) = a \wedge a \quad (A.12)$$

$$(a \vee \neg a) \wedge x = (\neg a \wedge a) \vee x \quad (A.13)$$

$$(\neg a \vee a) \wedge x = (a \wedge \neg a) \vee x \quad (A.14)$$

$$(a \wedge \neg a) \wedge x = a \wedge \neg a \quad (A.15)$$

$$(\neg a \wedge a) \wedge x = \neg a \wedge a \quad (A.16)$$

$$(x \wedge y) \vee (a \wedge \neg a) = (x \vee (a \wedge \neg a)) \wedge (y \vee (a \wedge \neg a)) \quad (A.17)$$

$$(x \wedge y) \vee (\neg a \wedge a) = (x \vee (\neg a \wedge a)) \wedge (y \vee (\neg a \wedge a)) \quad (A.18)$$

The system EqCSCL extends EqFSCL with the following axiom schemes, where a ranges over \mathcal{A} :

$$a \wedge (a \vee x) = a \quad (A.19)$$

$$a \vee (a \wedge x) = a \quad (A.20)$$

$$a \vee \neg a = a \vee T \quad (A.21)$$

$$a \wedge \neg a = a \wedge F \quad (A.22)$$

The system EqMSCL is based on EqFSCL but replaces axioms A.9 and A.10 with the following axioms:

$$x \wp (x \vee y) = x \tag{A.23}$$

$$x \wp (y \vee z) = (x \wp y) \vee (x \wp z) \tag{A.24}$$

$$(x \wp y) \vee (\neg x \wp z) = (x \vee z) \wp (\neg x \vee y) \tag{A.25}$$

$$(x \wp y) \vee (\neg x \wp z) = (\neg x \wp z) \vee (x \wp y) \tag{A.26}$$

$$((x \wp y) \vee (\neg x \wp z)) \wp u = (x \wp (y \wp u)) \vee (\neg x \wp (z \wp u)) \tag{A.27}$$

Finally, the system EqSSCL extends EqMSCL with one final axiom:

$$x \wp F = F \tag{A.28}$$

B. Proof of Proposition 3.9

Proposition B.1. *Let V be a valuation algebra. If V is memorizing and $a \in \mathcal{A}$ then*

$$a/(x \bullet a \bullet H) = a/H, \quad a \bullet x \bullet a \bullet H = x \bullet a \bullet H \quad (\forall H \in V) \quad (\text{B.1})$$

for all formulas x .

Proof. Let V be memorizing and let $a \in \mathcal{A}$. Because V is also contractive and repetition-proof, we find

$$a/(\mathsf{T} \bullet a \bullet H) = a/(a \bullet H) = a/H, \quad a \bullet \mathsf{T} \bullet a \bullet H = a \bullet a \bullet H = a \bullet H$$

for all $H \in V$, so T satisfies (B.1). Let $b \in \mathcal{A}$, then b satisfies (B.1) because V is memorizing. Suppose x, y are formulas that satisfy (B.1). Because $(\neg x) \bullet a \bullet H = x \bullet a \bullet H$ for all H , it immediately follows that $\neg x$ also satisfies (B.1). Additionally, because

$$\begin{aligned} a/(y \bullet x \bullet a \bullet H) &= a/(y \bullet (x \bullet a \bullet H)) \\ &= a/(y \bullet (a \bullet x \bullet a \bullet H)) \\ &= a/(y \bullet a \bullet (x \bullet a \bullet H)) \\ &= a/(x \bullet a \bullet H) \\ &= a/H \end{aligned}$$

and

$$\begin{aligned} a \bullet y \bullet x \bullet a \bullet H &= a \bullet y \bullet (x \bullet a \bullet H) \\ &= a \bullet y \bullet (a \bullet x \bullet a \bullet H) \\ &= a \bullet y \bullet a \bullet (x \bullet a \bullet H) \\ &= y \bullet a \bullet (x \bullet a \bullet H) \\ &= y \bullet (a \bullet x \bullet a \bullet H) \\ &= y \bullet (x \bullet a \bullet H) \\ &= y \bullet x \bullet a \bullet H, \end{aligned}$$

we find

$$\begin{aligned} a/((x \wp y) \bullet a \bullet H) &= \begin{cases} a/(x \bullet a \bullet H) & \text{if } x/(a \bullet H) = \mathsf{F} \\ a/(y \bullet x \bullet a \bullet H) & \text{otherwise} \end{cases} \\ &= \begin{cases} a/H & \text{if } x/(a \bullet H) = \mathsf{F} \\ a/H & \text{otherwise} \end{cases} \\ &= a/H \end{aligned}$$

and

$$\begin{aligned}
a \bullet (x \wedge y) \bullet a \bullet H &= \begin{cases} a \bullet x \bullet a \bullet H & \text{if } x/(a \bullet H) = F \\ a \bullet y \bullet x \bullet a \bullet H & \text{otherwise} \end{cases} \\
&= \begin{cases} x \bullet a \bullet H & \text{if } x/(a \bullet H) = F \\ y \bullet x \bullet a \bullet H & \text{otherwise} \end{cases} \\
&= (x \wedge y) \bullet a \bullet H.
\end{aligned}$$

This means also $x \wedge y$ satisfies (B.1). By induction, all formulas x satisfy (B.1). \square

Proposition B.2. *Let V be a valuation algebra. If V is static and $a \in \mathcal{A}$ then*

$$a/(x \bullet H) = a/H \quad (\forall H \in V) \quad (\text{B.2})$$

for all formulas x .

Proof. Let V be static and let $a \in \mathcal{A}$. Clearly, $a/(T \bullet H) = a/H$ for all $H \in V$, so T satisfies (B.2). Also, let $b \in \mathcal{A}$, then b satisfies (B.2) because V is static. Suppose x, y are formulas that satisfy (B.2). Then $a/((\neg x) \bullet H) = a/(x \bullet H) = a/H$, so $\neg x$ satisfies (B.2). Furthermore, because $a/(y \bullet (x \bullet H)) = a/(x \bullet H) = a/H$ we find

$$\begin{aligned}
a/((x \wedge y) \bullet H) &= \begin{cases} a/(x \bullet H) & \text{if } x/H = F \\ a/(y \bullet x \bullet H) & \text{otherwise} \end{cases} \\
&= \begin{cases} a/H & \text{if } x/H = F \\ a/H & \text{otherwise} \end{cases} \\
&= a/H.
\end{aligned}$$

and this means $x \wedge y$ satisfies (B.2). By induction, we are done. \square

Proposition (3.9a). *Let V be a valuation algebra. If V is memorizing then*

$$x/(y \bullet x \bullet H) = x/H, \quad x \bullet y \bullet x \bullet H = y \bullet x \bullet H \quad (\forall H \in V) \quad (\text{B.3})$$

for all formulas x, y .

Proof. Let V be memorizing and fix a formula y . We will prove this proposition by induction to the complexity of x . The T case is immediate. The a case for $a \in \mathcal{A}$ is already given by Proposition B.1.

The case $x = \neg x_1$ where x_1 satisfies (B.3): we can derive

$$\begin{aligned}
(\neg x_1)/(y \bullet (\neg x_1) \bullet H) & \quad (\neg x_1) \bullet y \bullet (\neg x_1) \bullet H \\
= \neg(x_1/(y \bullet x_1 \bullet H)) & \quad = x_1 \bullet y \bullet x_1 \bullet H \\
= \neg(x_1/H) & \quad = y \bullet x_1 \bullet H \\
= (\neg x_1)/H & \quad = y \bullet (\neg x_1) \bullet H
\end{aligned}$$

and therefore $\neg x_1$ also satisfies (B.3).

For the case $x = x_1 \wp x_2$ where x_1 and x_2 both satisfy (B.3), we will consider two possibilities separately:

Suppose $x_1/H = T$, then $(x_1 \wp x_2)/H = x_2/(x_1 \bullet H)$ and $(x_1 \wp x_2) \bullet H = x_2 \bullet x_1 \bullet H$. We will use a rewriting trick: if u and v are formulas and H is a valuation, then

$$u \bullet v \bullet H = ((v \vee T) \wp u) \bullet H.$$

Now we can derive

$$\begin{aligned} & (x_1 \wp x_2)/(y \bullet (x_1 \wp x_2) \bullet H) \\ &= \begin{cases} x_2/(x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H) & \text{if } x_1/(y \bullet x_2 \bullet x_1 \bullet H) = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2/(((y \vee T) \wp x_1) \bullet x_2 \bullet (x_1 \bullet H)) & \text{if } x_1/(((x_2 \vee T) \wp y) \bullet x_1 \bullet H) = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2/(x_1 \bullet H) & \text{if } x_1/H = T \\ F & \text{otherwise} \end{cases} \\ &= (x_1 \wp x_2)/H \end{aligned}$$

and

$$\begin{aligned} & (x_1 \wp x_2) \bullet y \bullet (x_1 \wp x_2) \bullet H \\ &= \begin{cases} x_2 \bullet x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H & \text{if } x_1/(y \bullet x_2 \bullet x_1 \bullet H) = T \\ x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2 \bullet x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H & \text{if } x_1/H = T \\ x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H & \text{otherwise} \end{cases} \\ &= x_2 \bullet x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H \\ &= x_2 \bullet ((y \vee T) \wp x_1) \bullet x_2 \bullet (x_1 \bullet H) \\ &= ((y \vee T) \wp x_1) \bullet x_2 \bullet (x_1 \bullet H) \\ &= x_1 \bullet y \bullet x_2 \bullet x_1 \bullet H \\ &= x_1 \bullet ((x_2 \vee T) \wp y) \bullet x_1 \bullet H \\ &= ((x_2 \vee T) \wp y) \bullet x_1 \bullet H \\ &= y \bullet x_2 \bullet x_1 \bullet H \\ &= y \bullet (x_1 \wp x_2) \bullet H \end{aligned}$$

so it satisfies (B.3).

Suppose otherwise, i.e. $x_1/H = F$, then $(x_1 \wp x_2)/H = F$ and $(x_1 \wp x_2) \bullet H = x_1 \bullet H$. We can derive

$$\begin{aligned} & (x_1 \wp x_2)/(y \bullet (x_1 \wp x_2) \bullet H) = (x_1 \wp x_2)/(y \bullet x_1 \bullet H) \\ &= \begin{cases} x_2/(y \bullet x_1 \bullet H) & \text{if } x_1/(y \bullet x_1 \bullet H) = T \\ F & \text{otherwise} \end{cases} \\ &= \begin{cases} x_2/(y \bullet x_1 \bullet H) & \text{if } x_1/H = T \\ F & \text{otherwise} \end{cases} \\ &= F \end{aligned}$$

and

$$\begin{aligned}
(x_1 \wp x_2) \bullet y \bullet (x_1 \wp x_2) \bullet H &= (x_1 \wp x_2) \bullet y \bullet x_1 \bullet H \\
&= \begin{cases} x_2 \bullet x_1 \bullet y \bullet x_1 \bullet H & \text{if } x_1/(y \bullet x_1 \bullet H) = \text{T} \\ x_1 \bullet y \bullet x_1 \bullet H & \text{otherwise} \end{cases} \\
&= \begin{cases} x_2 \bullet x_1 \bullet y \bullet x_1 \bullet H & \text{if } x_1/H = \text{T} \\ x_1 \bullet y \bullet x_1 \bullet H & \text{otherwise} \end{cases} \\
&= x_1 \bullet y \bullet x_1 \bullet H \\
&= y \bullet x_1 \bullet H \\
&= y \bullet (x_1 \wp x_2) \bullet H
\end{aligned}$$

so now it satisfies (B.3) as well.

Thus the case $x_1 \wp x_2$ also satisfies (B.3), which concludes our inductive proof. \square

Proposition (3.9b). *Let V be a valuation algebra. If V is static then*

$$x/(y \bullet H) = x/H \quad (\forall H \in V) \quad (\text{B.4})$$

for all formulas x, y .

Proof. Let V be static and fix a formula y . We will prove this proposition by induction to the complexity of x . The T case is immediate. The a case for $a \in \mathcal{A}$ is already given by Proposition B.2.

The case $x = \neg x_1$ where x_1 satisfies (B.4): we can derive

$$(\neg x_1)/(y \bullet H) = \neg(x_1/(y \bullet H)) = \neg(x_1/H) = (\neg x_1)/H$$

and therefore $\neg x_1$ also satisfies (B.4).

The case $x = x_1 \wp x_2$ where x_1 and x_2 both satisfy (B.4): we derive $x_1/(y \bullet H) = x_1/H$ and $x_2/(x_1 \bullet (y \bullet H)) = x_2/(y \bullet H) = x_2/H$, thus

$$\begin{aligned}
(x_1 \wp x_2)/(y \bullet H) &= \begin{cases} x_2/(x_1 \bullet y \bullet H) & \text{if } x_1/(y \bullet H) = \text{T} \\ \text{F} & \text{otherwise} \end{cases} \\
&= \begin{cases} x_2/H & \text{if } x_1/H = \text{T} \\ \text{F} & \text{otherwise} \end{cases} \\
&= (x_1 \wp x_2)/H
\end{aligned}$$

and therefore $x_1 \wp x_2$ also satisfies (B.4). \square